UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Povzporejanje metahevristik za NP-polne probleme

Rok Cvahte

Delo je pripravljeno v skladu s Pravilnikom o podeljevanju Prešernovih nagrad študentom, pod mentorstvom dr. Andreja Brodnika

Ljubljana, 2010

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Rok Cvahte,

z vpisno številko 63050056,

sem avtor diplomskega dela z naslovom:

Povzporejanje metahevristik za NP-polne probleme

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  dr. Andreja Brodnika
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki „Dela FRI“.

V Ljubljani, dne 22.02.2010 Podpis avtorja:

Zahvala

Mentorju bi se rad zahvalil za vse, kar me je tekom sodelovanja naučil, ter za njegov trud, pomoč in ideje, ki so pripomogli k nastanku te diplomske naloge.

Staršem bi se rad zahvalil, ker so mi študij omogočili in me ves čas podpirali, tako finančno kot tudi moralno. Najboljšemu prijatelju Filipu bi se rad zahvalil za brezpogojno podporo in številne nasvete, ki so naredili moj študij bolj uspešen.

Turboinštitutu in tamkajšnjemu vodji raziskav, dr. Andreju Lipeju, bi se rad zahvalil za omogočeno izvedbo preizkusa mojih algoritmov na superračunalniku LSC Adria. Andreju Krevlu iz Laboratorija za računalniške komunikacije bi se rad zahvalil za pomoč in potrpežljivost pri vzpostavitvi in upravljanju navideznih strojev, ki so omogočili del mojih preizkusov.

Dr. Borutu Robiču bi se rad zahvalil za njegovo pomoč pri pregledu slovarčka. Zahvala gre tudi dr. Marcu Chiarandiniju za predlaganje literature s področja metahevristik in njihovega povzporejanja.

Kazalo

Povzetek

V tem delu se bomo seznanili s praktičnimi problemi, za katere ne poznamo algoritmov, ki bi našli rešitev v manj kot eksponentnem času na deterministčnem Turingovem stroju. Ogledali si bomo pojem NP-polnosti, ki omogoča formalno razvrščanje problemov glede na njihovo zahtevnost. Sledi pregled štirih NP-ekvivalentnih problemov: problem največje neodvisne množice, problem največjega polnega podgrafa, problem najmanjšega pokritja grafa in problem trgovskega potnika.

Nadaljevali bomo z ogledom metahevrističnih algoritmov, ki nam omogočajo, da v času, praviloma bistveno krajšem od eksponentnega, dobimo suboptimalno rešitev določenega problema. Za zgled si bomo ogledali šest metahevrističnih algoritmov za reševanje problema največje neodvisne množice: požrešno iskanje, simulirano ohlajanje, razpršeno iskanje, metoda navzkrižne entropije, sistem mravelj in evolucijski algoritem.

Reševanje problemov lahko pospešimo tudi z uporabo več računalnikov. Tako si bomo ogledali orodja za pripravo vzporednih programov, in sicer si bomo podrobneje ogledali tehnologiji PVM in OpenMPI. Na podlagi zaporednih algoritmov in z uporabo OpenMPI in PVM bomo pripravili tri vzporedne metahevristične algoritme: simulirano ohlajanje, razpršeno iskanje in metoda navzkrižne entropije.

Sledi predstavitev rezultatov omenjenih zaporednih in vzporednih algoritmov, implementiranih v programskem jeziku C++, na štirih sistemih: dva različna strežnika Dell, heterogen sistem računalnikov in superračunalnik LSC Adria. Pri analizi bomo uporabili standardne DIMACS primerke.

Pri tem bomo videli, da se zaporedni algoritem simulirano ohlajanje obnese bolje tako od do sedaj najboljše znane implementacije kakor tudi od ostalih analiziranih algoritmov razpršeno iskanje in metoda navzkrižne entropije. Pri primerjavi zaporednih algoritmov z vzporednimi bomo videli, da prikazani vzporedni algoritmi običajno skrajšajo čas reševanja in izboljšajo kakovost rešitev, vendar je pohitritev ob enaki kakovosti precej slabša od teoretično pričakovane vrednosti. Sledila bo še primerjava okolij OpenMPI in PVM, kjer se izkaže, da se simulirano ohlajanje in razpršeno iskanje z uporabo PVM obneseta bistveno bolje kot pri uporabi OpenMPI.

Zaključili bomo s pregledom možnih smernic za nadaljnje delo.

Ključne besede:

kombinatorična optimizacija, metahevristični algoritmi, NP-polnost, vzporedni algoritmi

Abstract

In this work, we look at a class of very hard practical problems which, currently, can only be solved with algorithms running in exponential time on deterministic Turing machine. Further, we discuss the theory of NP-completeness, which allows us to classify problems based on their complexity. We proceed by looking at four NP-equivalent problems: maximum independent set problem, maximum clique problem, minimum vertex cover problem and traveling salesman problem.

We continue with a class of meta-heuristic algorithms, which provide suboptimal solutions – however, their running time is usually substantially smaller than exponential. We discuss six of such meta-heuristic algorithms: hill climbing, simulated annealing, scatter search, cross-entropy method, ant colony optimization and evolutionary algorithm.

To solve problems faster, we can deploy several computers simultaneously. We look at two technologies, allowing us to write parallel programs: OpenMPI and Parallel Virtual Machine (PVM). Based on serial algorithms and using OpenMPI and PVM, we build three parallel meta-heuristic algorithms: simulated annealing, scatter search and cross-entropy method.

Subsequently, we present test results of the mentioned serial and parallel algorithms, implemented in programming language C++ on four systems: two different Dell servers, heterogeneous system of computers and supercomputer LSC Adria. For the analysis, we use standard DIMACS instances.

Results show that serial simulated annealing algorithm outperforms currently best known implementation and it is also better than described cross-entropy method and scatter search. When we compare serial and parallel algorithms, we observe that parallel algorithms usually improve solving time and improve solution quality. However, the speedup – at the same quality – is substantially worse than the theoretically expected value. We also compare parallel environments OpenMPI and PVM and see that simulated annealing and scatter search algorithms using PVM perform considerably better than using OpenMPI.

We conclude with directions for possible future work.

Key words:

combinatorial optimization, meta-heuristic algorithms, NP-completeness, parallel algorithms

Slovarček

Slovarček je nastajal s pomočjo Islovarja [1] in slovarčka krajšav [2]. Določeni pojmi glede teorije NP-polnosti so povzeti po [3], glede vzporednih strojev in algoritmov po [4] ter glede metahevristik po [5].

Algoritem angl. *Algorithm*. Računalniški program, namenjen reševanju določenega formalno definiranega problema.

Čas izvajanja vzporednega algoritma Dejanski čas, ki preteče od začetka do konca izvajanja vzporednega algoritma, angl. *wall clock time*. Označimo ga s T_p, kjer p predstavlja število procesov.

Čas izvajanja zaporednega algoritma Analogen času izvajanja vzporednega algoritma. Označimo ga s T_1.

Deterministični Turingov stroj angl. *Deterministic Turing machine*, okr. *DTS*. Teoretični stroj, ki je po moči računanja ekvivalenten splošnonamenskim računalnikom.

Evolucijski algoritem angl. *Evolutionary algorithm*. Metahevristični algoritem, ki temelji na principih evolucije.

Funkcijski problem angl. *Function problem*. Problem, pri katerem želimo kot rezultat dobiti odgovor, ki je bolj kompleksen od preprostega „da“ ali „ne“, npr. zaporedje vozlišč na grafu.

InfiniBand Standard, ki določa hitro omrežje, namenjeno predvsem superračunalnikom.

Internetni protokol angl. *Internet protocol*, okr. *IP*. Protokol, namenjen prenosu podatkov preko omrežja paketnega preklapljanja.

Izostritev angl. *Intensification*. Princip metahevristik, ki je dopolnilen razpršitvi in govori o temeljitem preiskovanju določenega področja prostora rešitev z namenom odkrivanja čim boljših rešitev znotraj tega področja.

Jedro angl. *Core*. Del procesorja, ki izvaja enake naloge, kot jih sicer izvaja procesor pri enojedrnih procesorjih.

Kakovost rešitve angl. *Solution quality*. Definicija kakovosti rešitve je del definicije problema. V optimizacijskih problemih želimo doseči najboljšo možno.

Linearna pospešitev angl. *Linear speedup*. Pospešitev je linearna takrat, kadar raste linearno s številom procesov.

Lokalno iskanje angl. *Local search*. Ena izmed ključnih komponent metahevristik, ki govori o preiskovanju okolice rešitve.

Metahevristika angl. *Metaheuristic*. Metoda, ki uporablja hevristike za reševanje problemov kombinatorične optimizacije, za katere s trenutnimi računalniki in algoritmi ni mogoče preiskati celotnega prostora rešitev v zadovoljivo omejenem času. Običajno je rezultat metahevurističnih algoritmov suboptimalna rešitev.

Metoda navzkrižne entropije angl. *Cross-entropy method*. Metahevristični algoritem, ki temelji na simulaciji redkih dogodkov, angl. *rare event simulation*.

Naivno povzporejanje angl. *Naïve parallelization*. Povzporejanje, pri katerem se ne uporabi zapletenih metod povzporejanja, ampak se hkrati požene zaporedni algoritem na določenem številu procesov.

Nedeterministični Turingov stroj angl. *Non-deterministic Turing machine*, okr. *NTS*. Enak kot deterministični Turingov stroj, vendar razširjen z možnostjo alternativnih prehodov. Zanj velja, da kadar je na voljo več prehodov, stroj izbere pravi prehod.

Neodvisna množica angl. *Independent set*. Množica vozlišč v grafu, za katero velja, da med nobenim parom vozlišč ni povezave.

Nesprejemljiva rešitev angl. *Infeasible solution*. Rešitev določenega primerka, ki krši vsaj eno izmed omejitev problema.

NP problem angl. *NP problem*. Problem, ki ga lahko rešimo v polinomskem času na nedeterminističnem Turingovem stroju glede na velikost vhoda. Na determinističnem Turingovem stroju pa ni nujno, da ga je mogoče rešiti v polinomskem času.

NP-ekvivalenten problem angl. *NP-equivalent problem*. Funkcijski problem, ki je hkrati NP-težek in NP-lahek.

NP-lahek problem angl. *NP-easy problem*. Funkcijski problem, ki ga je mogoče rešiti v polinomskem času na deterministočnem Turingovem stroju, če imamo rešitev v polinomskem času za pripadajoči odločitveni problem.

NP-poln problem angl. *NP-complete*. NP problem, za katerega velja, da lahko nanj v polinomskem času prevedemo katerikoli NP problem.

NP-težek problem angl. *NP-hard*. Problem, na katerega je mogoče v polinomskem času prevesti neki NP-poln problem. NP-težki problemi so vsaj tako težki kot NP-polni.

Odločitveni problem angl. *Decision problem*. Problem, pri katerem moramo odgovoriti na vprašanje z „da“ ali „ne“.

Opravilo angl. *Task*. Enota izvajanja računalniškega programa.

Optimizacijski problem angl. *Optimization problem*. Problem, pri katerem moramo najti najboljšo možno rešitev glede na kakovost.

Pokritje grafa angl. *Vertex cover*. Množica vozlišč grafa, ki zadošča pogoju, da vsaka povezava grafa meji vsaj na eno vozlišče, vključeno v to množico.

Polinomska prevedba angl. *Polynomial-time reduction*. Prevedba določenega problema P_i na neki drug problem P_j, ki jo je mogoče izvesti v polinomskem času na deterministočnem Turingovem stroju.

Poln graf angl. *Complete graph*. Graf, za katerega velja, da je vsako vozlišče povezano z vsakim drugim vozliščem.

Pomnilnik z naključnim dostopom angl. *Random access memory*, okr. *RAM*. Pomnilnik, pri katerem dostop do podatkov na poljubnem naslovu in v poljubnem vrstnem redu traja enako dolgo.

Pospešitev angl. *Speedup*, okr. S_p. Razmerje med časom izvajanja zaporednega algoritma in časom izvajanja vzporednega algoritma, pri čemer vzporedni algoritem reši isti problem kot zaporedni algoritem, le da to stori na p procesih.

Povzporejanje angl. *Parallelization*. Proces, pri katerem se na podlagi zaporednega algoritma pripravi vzporedni algoritem.

Prenosni protokol angl. *Transmission control protocol*, okr. *TCP*. Eden izmed protokolov za prenos podatkov preko omrežja, v kombinaciji z IP protokolom.

Primerek angl. *Instance*. Dobimo ga iz problema, ko določimo vrednosti parametrov problema.

Primerjalni preizkus angl. *Benchmark* ali *Benchmark test*. Preizkus, ki je namenjen primerjavi različnih računalnikov z namenom objektivnega ugotavljanja razlike v sposobnostih (npr. v hitrosti obdelave).

Problem Definiramo ga kot neko splošno vprašanje, na katero želimo odgovoriti. Problem ima določene parametre, katerih vrednosti niso opredeljene, tj. parametri so prosti. Problem je definiran s splošnim opisom vseh njegovih parametrov in opisom, kakšne omejitve mora izpolnjevati rešitev problema.

Problem najmanjšega pokritja grafa angl. *Minimum vertex-cover problem*. Problem na grafu, pri katerem moramo najti najmanjše pokritje grafa z vozlišči.

Problem največje neodvisne množice angl. *Maximum independent set problem* ali *Maximum stable set problem*. Problem na grafu, pri katerem moramo najti največjo neodvisno množico vozlišč.

Problem največjega polnega podgrafa angl. *Maximum clique problem*. Problem na grafu, pri katerem moramo najti največji poln podgraf.

Problem trgovskega potnika angl. *Traveling salesman problem*, okr. *TSP*. Problem na grafu, pri katerem je potrebno najti najkrajši možni obhod čez vsa vozlišča, pri čemer mora biti vsako vozlišče obiskano natanko enkrat. Povezave med vozlišči so utežene in predstavljajo razdaljo med vozlišči. Ločimo simetrični in asimetrični problem trgovskega potnika.

Proces angl. *Process*. Proces je entiteta, ki izvaja določeno opravilo (ponavadi vzporednega) algoritma. Potrebno je razlikovati med procesom in procesnim elementom, saj lahko na določenem procesnem elementu teče več procesov hkrati (vendar se običajno teži k razmerju $1 : 1$).

Procesni element angl. *Processing element*. Element, na katerem teče izvajanje določenega procesa, ponavadi enega. V primeru enojedrnega procesorja procesni element enačimo s procesorjem. V primeru večjedrnih procesorjev procesni element enačimo z jedrom procesorja.

Procesor angl. *Processor*. Del računalnika, ki razpoznava in izvaja ukaze, zapisane kot računalniški program v strojnem jeziku. Procesor ima vsaj

eno jedro (označimo kot enojedrni procesor), lahko pa ima tudi več kot eno jedro (dvojedrni, štirijedrni itd. procesor). Pri izvajanju ukazov ima procesor na voljo določeno količino pomnilnika z naključnim dostopom.

Programski vmesnik angl. *Application programming interface*, okr. *vmesnik*. Način, s katerim je mogoče na preprost način določiti, katere vmesnike mora nuditi določen program oziroma orodje.

Prostor rešitev angl. *Solution space*. Za določen primerek označimo kot prostor rešitev vse možne rešitve (tj. naredimo vse možne kombinacije vseh vrednosti vsake spremenljivke primerka).

Psevdokoda angl. *Pseudocode*. Način predstavitve algoritma na visokem nivoju, ki je neodvisen od specifičnega programskega jezika in je namenjen izključno tolmačenju algoritma.

Računalnik angl. *Computer*. Naprava, namenjena samodejnemu izvajanju računalniških programov, s katerimi se obdelujejo in shranjujejo podatki. Vsebuje vsaj en procesor in določeno količino pomnilnika z naključnim dostopom.

Razpošiljanje angl. *Broadcast*. Komunikacijska operacija, pri kateri en proces pošlje podatke vsem ostalim procesom.

Razpršeno iskanje angl. *Scatter search*. Metahevristični algoritem, ki izhaja iz težnje po čim večji preiskanosti prostora rešitev.

Razpršitev angl. *Diversification*. Princip metahevristik, ki je dopolnilen izostritvi in govori o temeljitem preiskovanju prostora rešitev z namenom razpoznavanja področij, v katerih je kakovost rešitev boljša.

Simulirano ohlajanje angl. *Simulated annealing*. Metahevristični algoritem, ki temelji na metalurškem postopku ohlajanja kovin.

Sistem mravelj angl. *Ant system* ali *Ant colony optimization*. Metahevristični algoritem, ki črpa iz biologije in simulira obnašanje mravelj.

Sprejemljiva rešitev angl. *Feasible solution*. Rešitev določenega primerka, ki ne krši nobene izmed omejitev problema.

Strošek režije angl. *Overhead cost*. Strošek, ki ni neposredno del reševanja določenega problema, ampak določene vrste režije (npr. komunikacija pri vzporednem algoritmu).

Superračunalnik angl. *Supercomputer*. Računalnik, ki je bistveno bolj zmogljiv od osebnega računalnika, v smislu hitrosti in količine obdelanih podatkov.

Topologija omrežja angl. *Network topology*. Fizična ali logična razporeditev elementov omrežja.

Učinkovitost angl. *Efficiency*, okr. E_p. Pri ocenjevanju vzporednih algoritmov razmerje med pospešitvijo in številom procesov p.

Uporabniški datagramski protokol angl. *User datagram protocol*, okr. *UDP*. Eden izmed protokolov za prenos podatkov preko omrežja, v kombinaciji z IP protokolom.

Uteženi graf angl. *Weighted graph*. Graf, v katerem ima vsaka izmed povezav pripisano težo.

Večmodalna funkcija angl. *Multimodal function*. Funkcija, ki ima več lokalnih optimumov in se lahko uporablja za ocenjevanje uspešnosti optimizacijskih algoritmov.

Zrnatost angl. *Granulation*. Lastnost orodja za vzporedno računanje z vidika pogostosti komunikacije med procesi in neodvisnosti posameznih procesov.

Poglavje 1

Uvod

V matematiki poznamo kar precej težkih problemov, ki zahtevajo veliko časa za izračun rešitve. Ker želimo biti pri ocenjevanju zahtevnosti problemov konkretni in se ne želimo opirati na relativne pojme, kot so lahek problem, težek problem ipd., se bomo v poglavju 2 seznanili s teorijo NP-polnosti, ki je bila leta 1979 natančno opisana v [3] in govori o tem, kako težki so določeni odločitveni in funkcijski problemi. Pri tem se opira na točno definirane pojme, kot so P, NP, NP-težek, NP-poln itd. problem. S temi pojmi je bila v [3] točno opredeljena zahtevnost reševanja raznovrstnih odločitvenih in funkcijskih problemov. Obstaja nezanemarljiv delež problemov iz resničnega sveta, za katere poznamo le algoritme z eksponentno časovno zahtevnostjo v odvisnosti od velikosti parametrov, na deterministočnem Turingovem stroju (DTS). V praksi to pomeni, da je izračun rešitev časovno zelo zahteven in pogosto nesprejemljivo dolg. Da bi z abstraktnega prešli na konkretno, si bomo pogledali štiri dejanske probleme: problem največje neodvisne množice, problem največjega polnega podgrafa, problem najmanjšega pokritja grafa in problem trgovskega potnika. Za reševanje teh problemov poznamo trenutno le algoritme, ki imajo eksponentno časovno zahtevnost na DTS.

Kot posledica ugotovitev prejšnjega odstavka nas zanima, ali je mogoče reševati omenjene probleme na način, pri katerem najdemo rešitev hitreje kot v eksponentnem času. Kolikor nam zadoščajo rešitve, ki v splošnem niso optimalne, si lahko pomagamo z metahevrističnimi algoritmi, ki si jih bomo ogledali v poglavju 3. Njihova značilnost je, da najdejo rešitve problema v času, ki je krajši od eksponentnega, vendar pri tem običajno ne najdejo optimalne rešitve. Pogledali si bomo šest različnih metahevrističnih algoritmov: požrešno iskanje, simulirano ohlajanje, razpršeno iskanje, metoda navzkrižne entropije, sistem mravelj in evolucijski algoritem.

Ko začnemo delati z vzporednimi algoritmi, je zelo pomembno, kakšna orodja imamo na razpolago za izkoriščanje določenega vzporednega stroja. V splošnem jih ločimo glede na nivo zrnatosti. Pri fini zrnatosti so med bolj znanimi MPI, OpenMP in PVM ter pri grobi zrnatosti računalništvo na mreži, angl. *grid computing*, in računalništvo v oblaku, angl. *cloud computing*. Vmesnik MPI ima kar nekaj različnih implementacij, med njimi tudi OpenMPI. Ena izmed pomembnejših implementacij računalništva na mreži je Globus Toolkit. V poglavju 4 si bomo natančno ogledali in primerjali OpenMPI ter PVM.

Ker si želimo dosegati čim boljše rešitve s pomočjo metahevrističnih algoritmov, in to v čim krajšem času, je zanimiva ideja povzporejanja metahevrističnih algoritmov, s katero se bomo seznanili v poglavju 5. Pri povzporejanju kot osnovo vzamemo zaporedno različico določenega metahevrističnega algoritma in pripravimo vzporedno različico, ki se lahko izvaja na več kot enem računalniku hkrati. Pri tem želimo pripraviti čim bolj učinkovit vzporedni algoritem, ki bo razpoložljiv vzporedni sistem čim bolje obremenil. Ogledali si bomo pripravo vzporednih različic algoritmov simulirano ohlajanje, razpršeno iskanje in metoda razpršene entropije.

Pripravljene algoritme želimo med seboj empirično primerjati, zato si bomo v poglavju 6 ogledali vzporedne stroje, na katerih so bili izvedeni preizkusi. Za kakovostno analizo rezultatov preizkusov je pomembno izvesti preizkuse pod kontroliranimi pogoji in na natančno določeni strojni in programski opremi, zato si bomo ogledali specifikacije vseh sistemov. Poleg tega se bomo seznanili tudi z načinom preizkušanja z vidika večkratnih meritev in izbire različnih primerkov.

V poglavju 7 si bomo pogledali in primerjali rezultate preizkusov implementiranih zaporednih in vzporednih algoritmov za reševanje problema največje neodvisne množice. Zaključke diplomske naloge bomo zapisali v poglavju 8, kjer si bomo ogledali tudi možne smernice za nadaljnje delo.

Poglavje 2

NP-polnost

V tem poglavju si bomo ogledali potreben del teorije NP-polnosti, kot je bila leta 1979 natančno opisana v [3]. Najprej si bomo pogledali deterministični Turingov stroj in razred problemov P. Sledil bo opis nedeterminističnega Turingovega stroja in razredov NP. Nato si bomo ogledali razmerje med razredoma P in NP ter poglavje zaključili z ogledom štirih konkretnih problemov na grafih.

2.1 Deterministični Turingov stroj

Deterministični Turingov stroj (DTS) [3], angl. *deterministic Turing machine*, je definiran z dvosmernim neomejenim trakom, bralno-pisalno glavo in nadzorno enoto. **Trak** je razdeljen na celice in v vsaki izmed njih je zapisan simbol iz končne abecede ali poseben simbol B, ki označuje, da je ta celica prazna. **Bralno-pisalna glava**, v nadaljevanju **glava**, vidi natanko en simbol na traku. Mogoče jo je premikati levo/desno po en znak hkrati. **Nadzorna enota** vsebuje končno mnogo stanj in je v vsakem trenutku v enem izmed teh stanj. S pomočjo bralno-pisalne glave lahko iz traku prebere simbol in na njegovo mesto zapiše drug ali enak simbol.

Nadzorna enota deluje tako, da v vsakem koraku na podlagi simbola, ki je bil prebran, in trenutnega stanja zapiše nov simbol na trak, glavo premakne v levo ali desno in preide v novo stanje.

Deterministični Turningov stroj formalno definiramo kot sedmerko $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, kjer Q predstavlja končno množico stanj nadzorne enote, Σ predstavlja končno množico vhodnih simbolov, Γ predstavlja končno množico tračnih simbolov, δ predstavlja funkcijo prehodov, q_0 predstavlja začetno stanje, B predstavlja prazen simbol in F predstavlja množico končnih stanj nadzorne enote. Pri tem veljajo naslednje omejitve: $\Sigma \subset \Gamma$, $q_0 \in Q$, $B \in \Gamma$ in

$F \subseteq Q$. Funkcija prehodov δ je definirana kot $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, D\}$, kjer L in D predstavljata premik glave za en simbol v levo oziroma desno.

Vsak Turingov stroj ima svoj jezik in sprejme besede, ki so v tem jeziku. To, da Turingov stroj sprejme določeno besedo, pomeni, da nadzorna enota v končnem številu korakov pride iz začetnega stanja q_0 v eno izmed končnih stanj F. Pri tem je potrebno pred začetkom zagona Turingovega stroja na trak zapisati vhodno besedo (na mesta od 1 do n, kjer je n dolžina vhodne besede). Jezik Turingovega stroja M formalno definiramo kot $L(M) = \{w | w \in \Sigma^* \wedge M \textit{ sprejme } w\}$.

Ker želimo DTS uporabljati za reševanje problemov, moramo definirati še relacijo med jeziki in problemi. Vsak primerek problema lahko na določen način zapišemo kot zaporedje vhodnih simbolov Σ^*. Pri takšni predstavitvi ločimo med zaporedji vhodnih simbolov, ki ne predstavljajo nobenega primerka problema, in zaporedji, ki predstavljajo primerek problema. Slednja zaporedja dalje delimo na tista, za katera DTS odgovori z „da“, in tista, za katera odgovori z „ne“.

2.2 Razred P

Razred P obsega probleme, za katere poznamo programe, ki imajo polinomsko časovno zahtevnost na DTS. Formalno razred P definiramo kot $\mathrm{P} = \{L | \textit{obstaja program s polinomsko časovno zahtevnostjo na DTS in velja } L = L(M)\}$ [3].

Polinomska časovna zahtevnost programa je definirana kot omejitev časa izvajanja programa od zgoraj z velikostjo strukture, s katero je podan primerek problema. Pri tem mora biti struktura smiselna in ne sme vsebovati odvečnih informacij.

2.3 Nedeterministični Turingov stroj

Nedeterministični Turingov stroj (NTS) [3], angl. *non-deterministic Turing machine*, je podoben DTS, z razliko, da je funkcija prehodov δ definirana drugače. Za določeno kombinacijo simbola pod glavo in stanja nadzorne enote lahko definiramo več alternativnih prehodov.

Pri izvajanju NTS izbere tisti prehod, da bo NTS sprejel vhodno besedo, kolikor le-ta pripada jeziku NTS. V praksi NTS ni mogoče izdelati. Izbiro pravega prehoda si lahko predstavljamo na naslednji način. Na voljo imamo magični kovanec, ki pove, katerega izmed možnih prehodov je potrebno izbrati, da bo NTS sprejel vhodno besedo, kolikor le-ta pripada jeziku NTS. Na DTS

je mogoče NTS simulirati tako, da na vsakem koraku, kjer imamo več kot en alternativni prehod, sledimo vsem prehodom. Na ta način se razvije drevesna struktura. V tem primeru DTS sprejme vhodno besedo, če je uspešna vsaj ena veja drevesa.

2.4 Razred NP

Razred NP zajema probleme, za katere poznamo programe, ki imajo polinomsko časovno zahtevnost na NTS. Formalno definiramo razred NP kot NP $= \{L | \textit{obstaja program s polinomsko časovno zahtevnostjo na NTS in velja } L = L(M)\}$ [3].

Razred NP vsebuje poseben podrazred z imenom **NP-poln** [3], označimo NP-poln $\subset$ NP. Za vsak problem, ki pripada razredu NP-polnih problemov, velja, da je mogoče v polinomskem času nanj prevesti katerega koli izmed problemov v razredu NP. Za vsak problem, ki sodi v razred NP-poln, pravimo, da je **NP-poln**.

Dokaz za obstoj „prvega“ NP-polnega problema (problem izpolnjivosti) je leta 1971 objavil Cook [6]. Po tem je bilo bolj enostavno pokazati za določene druge probleme razreda NP, da so tudi ti NP-polni. Če želimo za določen problem P_i, ki pripada razredu NP, dokazati, da je tudi NP-poln, zadošča dokaz, da je mogoče v polinomskem času prevesti vsaj enega izmed NP-polnih problemov P_j na problem P_i.

Poleg razredov NP in NP-poln obstaja tudi razred **NP-težek** [3], angl. *NP-hard*. V razred NP-težkih problemov sodijo vsi problemi, za katere velja, da je mogoče nanje v polinomskem času prevesti katerega koli izmed NP problemov. Drugače povedano, določen problem P_i je NP-težek natanko tedaj, ko obstaja NP-poln problem P_j in je mogoče problem P_j v polinomskem času prevesti na problem P_i. Vsak izmed NP-težkih problemov je vsaj tako zahteven kot vsak izmed NP problemov. Izpostaviti je potrebno tudi to, da NP-težki problemi, ki so izven množice NP, v primeru enakosti P = NP ne postanejo rešljivi v polinomskem času.

Kadar obravnavamo funkcijske probleme (npr. problem trgovskega potnika) v okviru NP-polnosti, moramo definirati še dva razreda. Problem P_i pripada razredu **NP-lahek**, angl. *NP-easy*, natanko tedaj, ko obstaja prevedba v polinomskem času problema P_i na neki NP problem P_j. Neki drug problem P_k pripada razredu **NP-ekvivalenten**, angl. *NP-equivalent*, če je problem P_k hkrati NP-lahek in NP-težek [3]. Razred NP-ekvivalenten je analogen razredu NP-polnih problemov za funkcijske probleme.

2.5 Razmerje med razredoma P in NP

Razred P je vsebovan v razredu NP, označimo P $\subseteq$ NP. To velja, ker lahko vsak program za reševanje problema, ki pripada razredu P in se izvaja na DTS, preprosto pretvorimo na program za reševanje problema razreda NP, ki se izvaja na NTS. Pretvorba je trivialna, saj NTS omogoča vse, kar omogoča DTS. Pri tem seveda ne uporabimo ene izmed možnosti, ki jih ponuja NTS, tj. definiranje funkcije δ z alternativnimi prehodi za določeno kombinacijo simbola pod glavo in stanja nadzorne enote.

Kakšno je razmerje v obratni smeri, iz razreda NP v razred P, ne vemo. Trenutno ne poznamo načina, da bi program, ki se v polinomskem času izvede na NTS, pretvorili v program, ki se v polinomskem času izvede na DTS. Prav tako nimamo dokaza, ki bi govoril o tem, da to ni izvedljivo. V prvem primeru, če je mogoče pretvoriti program z NTS na DTS in program pri tem ohrani polinomsko časovno zahtevnost, velja enakost P = NP. V nasprotnem primeru velja stroga vsebovanost P $\subset$ NP.

Kljub mnogim poskusom raziskovalcev v preteklih desetletjih odgovora na to vprašanje še ni [7]. Obstajajo ugibanja na podlagi empiričnih podatkov o tem, katera izmed izključujočih se variant drži. Večina vodilnih znanstvenikov v današnjih dneh je mnenja, da velja stroga vsebovanost P $\subset$ NP [7]. Opozoriti gre, da je mogoče v kriptografiji najti kar nekaj primerov, ki se zanašajo na predpostavko P $\subset$ NP. Moč in varnost določenih algoritmov v kriptografiji izvira iz tega, da je potrebno izredno veliko časa za razbitje določene kode, ker je problem razbitja kode NP-poln ali NP-ekvivalenten [8]. Kolikor bi se izkazalo, da velja P = NP ali da je mogoče izdelati sistem, ki v bistveno krajšem času kot doslej reši NP-poln problem (kot na primer kvantni ali DNK računalnik), bi to imelo posledice za določene kriptirane komunikacije v preteklosti, ki so bile morda prestrežene in so še nerazbite.

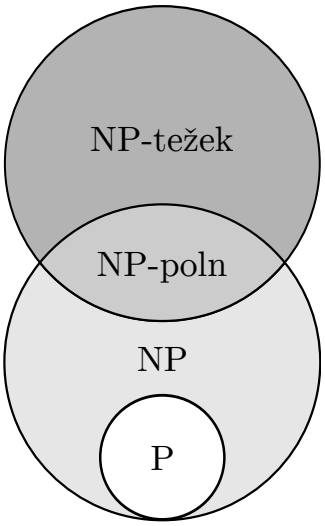
Razmerje med razredi P, NP, NP-poln in NP-težek je prikazano na sliki 2.1.

2.6 Primeri NP-ekvivalentnih problemov

V tem podpoglavju bomo pogledali štiri NP-ekvivalentne probleme: problem največje neodvisne množice, problem največjega polnega podgrafa, problem najmanjšega pokritja grafa in problem trgovskega potnika. Ker vsi štirje navedeni problemi temeljijo na grafu, bomo graf najprej formalno definirali.

Graf $G = (V, E)$ je sestavljen iz množice vozlišč V in množice povezav med vozlišči E. Množica vozlišč vsebuje n vozlišč in je definirana kot $V =$

Slika 2.1: Razredi kompleksnosti



$\{v_1, v_2, \ldots v_n\}$. Množica povezav med vozlišči vsebuje m parov vozlišč in je definirana kot $E \subseteq \{\{v_i, v_j\} : v_i, v_j \in V \wedge v_i \neq v_j\}$.

Grafe lahko delimo na usmerjene in neusmerjene. O usmerjenih grafih govorimo, kadar ločimo med povezavama $\{v_i, v_j\}$ in $\{v_j, v_i\}$. V nasprotnem primeru, ko ne ločimo med povezavama, govorimo o neusmerjenih grafih.

Vsaki izmed povezav v grafu lahko priredimo težo, zapišemo $\forall\{v_i, v_j\} \in E : W(v_i, v_j) \in \mathcal{R}$. V takšnem primeru govorimo o uteženem grafu, angl. *weighted graph*.

Pri predstavitvi naslednjih štirih problemov bomo z S označevali množico vseh možnih rešitev. Pri problemih največje neodvisne podmnožice, največjega polnega podgrafa in najmanjšega pokritja grafa govorimo o delitvi vozlišč grafa v dve disjunktni podmnožici (vključena in izključena vozlišča), in zato je vseh možnih rešitev 2^n. Pri problemu trgovskega potnika govorimo o zaporedju vseh vozlišč brez ponavljanja, invariantno na obratni vrstni red, in je zato vseh možnih rešitev $n!/2$.

Reševanje problema največje neodvisne podmnožice in njemu enakovrednih problemov se v praksi uporablja za minimizacijo Boolovih izrazov, angl. *Boolean logic minimization*, v teoriji kod za odpravljanje napak, angl. *error-correcting code*, pri zaznavanju tekmovanja za vire, angl. *race condition detection*, itd. Problem trgovskega potnika je v praksi uporaben za različne situacije, v katerih imamo določene lokacije (vozlišča) in povezave med njimi ter nas zanima, kako na najcenejši način obiščemo vse lokacije, npr. priprava

poti avtobusov, poti dostave blaga s tovornjaki trgovinam, poti dostave blaga z ladjami v pristanišča, poti servisnih tehnikov med lokacijami.

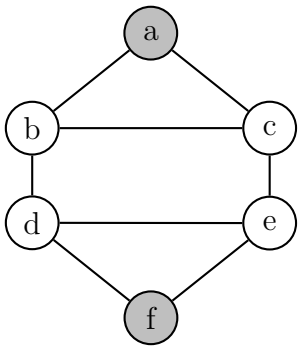
2.6.1 Problem največje neodvisne množice

Angl. *Maximum stable set problem*. Na grafu $G = (V, E)$ za neodvisno množico vozlišč $Z \subseteq V$ velja, da nobeno izmed vozlišč ni sosednje nobenemu drugemu vozlišču v množici Z. Zapišemo kot $Z = \{v_1, v_2, \dots v_k\}$ in velja $\forall v_i \forall v_j \in Z : \{v_i, v_j\} \notin E$.

Odločitveni problem je definiran kot preveritev, ali za celo število K, $0 \leq K \leq |V|$, na grafu G obstaja neodvisna podmnožica vozlišč $Z \subseteq V$, da velja $|Z| \geq K$. Ta odločitveni problem je NP-poln [3]. Pri optimizacijskem problemu največje neodvisne množice želimo najti neodvisno množico Z_b, ki ima največje možno število vozlišč. To zapišemo kot $\forall Z_i \in S : |Z_i| \leq |Z_b|$. Ta optimizacijski problem je NP-ekvivalenten [3].

Primer problema in ena izmed možnih rešitev sta prikazana na sliki 2.2, rešitev je množica $\{a, f\}$. Pri tem ne obstaja rešitev, ki bi vsebovala več kot dve vozlišči, zato je prikazana rešitev največja možna.

Slika 2.2: Primer največje neodvisne množice



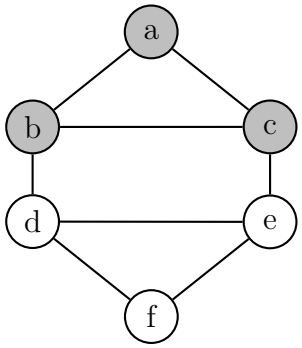
2.6.2 Problem največjega polnega podgrafa

Angl. *Maximum clique problem*. Na grafu $G = (V, E)$ je klika definirana kot podmnožica vozlišč $Z \subseteq V$. Pri tem mora veljati, da je vsako vozlišče podmnožice Z na grafu G povezano z vsakim drugim vozliščem iz te podmnožice. Zapišemo kot $Z = \{v_1, v_2, \dots v_k\}$ in velja $\forall v_i \forall v_j \in Z \wedge i \neq j : \{v_i, v_j\} \in E$. Vsaka klika enolično določa poln podgraf grafa G, označimo ga s H in velja $H = (Z, F)$, kjer $F = \{\{u_i, u_j\} : u_i, u_j \in Z \wedge i \neq j\}$.

Odločitveni problem je definiran kot preveritev, ali za celo število K, $0 \leq K \leq |V|$, na grafu G obstaja klika $Z \subseteq V$, da velja $|Z| \geq K$. Ta odločitveni problem je NP-poln [9]. Pri optimizacijskem problemu največjega polnega podgrafa želimo najti takšno kliko Z_b, ki ima največje možno število vozlišč. To zapišemo kot $\forall Z_i \in S : |Z_i| \leq |Z_b|$. Ta optimizacijski problem je NP-ekvivalenten [3].

Primer problema in ena izmed možnih rešitev sta prikazana na sliki 2.3, rešitev je množica $\{a, b, c\}$. V primeru ne obstaja rešitev, ki bi vsebovala več kot tri vozlišča, zato je prikazana rešitev največja možna.

Slika 2.3: Primer največjega polnega podgrafa



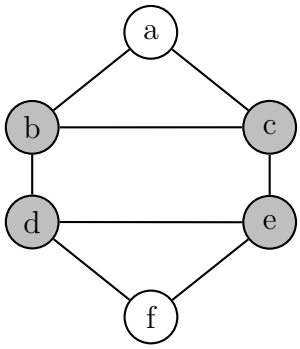
2.6.3 Problem najmanjšega pokritja grafa

Angl. *Minumum vertex cover problem*. Na grafu $G = (V, E)$ je pokritje grafa definirano kot množica vozlišč $Z \subseteq V$, tako da je za vsako povezavo na grafu vsaj eno izmed vozlišč vključeno v pokritje grafa. Zapišemo kot $Z = \{v_1, v_2, \ldots v_k\}$ in velja $\forall \{u_i, u_j\} \in E : u_i \in Z \vee u_j \in Z$.

Odločitveni problem je definiran kot preveritev, ali za celo število K, $0 \leq K \leq |V|$, na grafu G obstaja pokritje vozlišč $Z \subseteq V$, da velja $|Z| \leq K$. Ta odločitveni problem je NP-poln [9]. Pri optimizacijskem problemu najmanjšega pokritja grafa je cilj najti pokritje grafa Z_b, ki ima najmanjše možno število vozlišč. To zapišemo podobno kot pri prejšnjih dveh problemih, $\forall Z_i \in S : |Z_b| \leq |Z_i|$. Ta optimizacijski problem je NP-ekvivalenten [3].

Primer problema in ena izmed možnih rešitev sta prikazana na sliki 2.4, rešitev je množica $\{b, c, d, e\}$. Pri tem ne obstaja rešitev, ki bi vsebovala manj kot štiri vozlišča, zato je prikazana rešitev najmanjša možna.

Slika 2.4: Primer najmanjšega pokritja grafa



2.6.4 Enakost reševanja problemov

Pri reševanju naslednjih treh odločitvenih problemov, problema neodvisne množice, problema polnega podgrafa in problema najmanjšega pokritja grafa, v bistvu rešujemo isti problem. Za poljuben graf $G = (V, E)$ in podmnožico vozlišč $Z \subseteq V$ so naslednje tri izjave ekvivalentne [3]:

1. Z je pokritje grafa G,
2. $V \backslash Z$ je neodvisna podmnožica grafa G,
3. $V \backslash Z$ je klika komplementarnega grafa G^c, kjer velja $G^c = (V, E^c)$, pri čemer $E^c = \{\{v_i, v_j\} : v_i, v_j \in V \wedge \{v_i, v_j\} \notin E\}$.

Odločitveni problem pokritja grafa je NP-poln problem, dokazano s prevedbo problema 3SAT [3]. Zaradi zgoraj navedene ekvivalence sta tudi odločitvena problema neodvisne podmnožice in polnega podgrafa NP-polna.

2.6.5 Problem trgovskega potnika

Angl. *Traveling salesman problem*. Na grafu $G = (V, E)$, ki vsebuje n vozlišč, je Hamiltonov cikel definiran kot zaporedje vozlišč $T = \{v_1, v_2, \ldots, v_n\}$, kjer $v_i \in V \wedge 1 \leq i \leq n$. Pri tem je vsako izmed vozlišč obiskano natanko enkrat in obstaja povezava med dvema zaporednima vozliščema ter med začetnim in končnim vozliščem. Zapišemo kot $\{v_i, v_{i+1}\} \in T \wedge 1 \leq i < n : \{v_i, v_{i+1}\} \in E$ in $\{v_n, v_1\} \in E$. Ceno takšnega obhoda grafa definiramo kot $C(T) = W(\{v_n, v_1\}) + \sum_{i=1}^{n-1} W(\{v_i, v_{i+1}\})$, pri čemer $W(\{v_i, v_j\})$ označuje ceno povezave $\{v_i, v_j\}$.

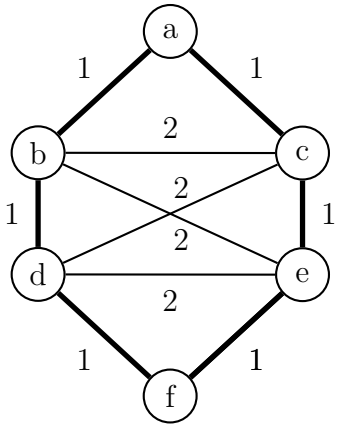
Odločitveni problem trgovskega potnika definiramo kot preveritev, ali za število $K \in \mathcal{Z}_0^+$ na grafu G obstaja Hamiltonov cikel T, da velja $C(T) \leq K$. Ta

odločitveni problem je NP-poln [3]. Pri optimizacijskem problemu trgovskega potnika želimo najti zaporedje vozlišč T_b, katerega cena je najmanjša možna. To zapišemo kot $\forall T_i \in S : C(T_b) \leq C(T_i)$. Ta optimizacijski problem je NP-ekvivalenten [3].

Glede na to, ali je problem trgovskega potnika postavljen na usmerjenem ali na neusmerjenem grafu, ločimo dve vrsti, simetrični in asimetrični problem. Pri simetričnem je teža povezave $\{v_i, v_j\}$ enaka teži povezave $\{v_j, v_i\}$, to zapišemo kot $\forall\{v_i, v_j\} \in E : W(\{v_i, v_j\}) = W(\{v_j, v_i\})$. Pri asimetričnem problemu teža povezave $\{v_i, v_j\}$ ni nujno enaka teži povezave $\{v_j, v_i\}$.

Na sliki 2.5 je prikazan preprost problem trgovskega potnika, simetrične vrste, katerega najboljša rešitev je zaporedje vozlišč $\{a, b, d, f, e, c\}$, ki ima ceno 6.

Slika 2.5: Primer problema trgovskega potnika



Poglavje 3

Kombinatorična optimizacija

Kombinatorična optimizacija, angl. *combinatorial optimization*, je področje, ki se ukvarja z iskanjem rešitev v množici možnih rešitev [5, 10]. Naštejmo nekaj problemov, ki spadajo v področje kombinatorične optimizacije:

- celoštevilsko programiranje (angl. *integer programming*),
- iskanje minimalnega vpetega drevesa (angl. *minimum spanning tree*),
- linearno programiranje (angl. *linear programming*),
- problem nahrbtnika (angl. *knapsack problem*).
- problem najkrajše poti na grafu (angl. *shortest path problem*),
- problem najmanjšega pokritja grafa (angl. *minimum vertex cover*),
- problem največje neodvisne množice (angl. *maximum independent set problem*),
- problem največjega polnega podgrafa (angl. *maximum clique problem*),
- problem n dam (angl. *n-queens puzzle*),
- problem trgovskega potnika (angl. *traveling salesman problem*),
- problem voznega reda (angl. *vehicle routing problem*).

Določeni problemi kombinatorične optimizacije sodijo v razred P in poznamo algoritme, ki najdejo optimalno rešitev v polinomskem času na DTS (od zgoraj naštetih problem najkrajše poti na grafu in problem n dam). Vendar za večino problemov, ki sodijo v kombinatorično optimizacijo, velja, da pripadajo razredu NP-ekvivalentnih problemov. Zaradi tega se je skozi leta razvil nabor algoritmov, ki pridejo do rešitve problema v času, ki je manjši od eksponentnega, vendar je v splošnem ta rešitev suboptimalna. Pri tem obstaja težnja, da se z izboljševanjem algoritmov čim bolj približamo optimalni rešitvi.

Izpostaviti moramo, da v splošnem ni mogoče vedeti, katera rešitev je optimalna, dokler ni bil preiskan celoten prostor rešitev S (v določenih posebnih primerih je to mogoče že prej, npr. z izločanjem delov prostora rešitev). Zaradi tega za rešitev, ki je bila dobljena s pomočjo približnega algoritma, v splošnem ne vemo, ali je optimalna ali ne. Vsak problem ima definiran svoj prostor rešitev S, ki vsebuje m rešitev S_i, $1 \leq i \leq m$.

Kadar govorimo o problemih optimizacije, moramo med drugim določiti tudi, kaj optimiziramo. V kombinatorični optimizaciji optimiziramo **kakovost rešitve**, angl. *fitness function* ali *objective function*. Definicija kakovosti rešitve je del definicije problema (npr. pri problemu neodvisne podmnožice grafa kot kakovost rešitve definiramo število vključenih vozlišč). Za neko rešitev S_i kakovost rešitve označimo z $f(S_i) \in \mathcal{R}$.

3.1 Metahevristike

Z nazivom metahevristike označujemo skupino algoritmov, ki so v prvi vrsti namenjeni reševanju optimizacijskih problemov, ki ne sodijo v razred P [5, 10]. Metahevristični algoritmi običajno najdejo eno ali več suboptimalnih rešitev v času, ki je manjši od eksponentnega. V tuji literaturi so metahevristični algoritmi poznani tudi pod imenom angl. *stochastic local search* [10]. V splošnem velja, da več časa kot ima metahevristični algoritem na voljo, boljša bo kakovost rešitve (angl. *anytime algorithm*).

Ena izmed glavnih idej metahevristik je **soseščina** [5, 10], angl. *neighborhood*. Če imamo določeno rešitev S, njeno soseščino definiramo s pomočjo poljubne funkcije $N(S_M)$, ki vrne množico sosednjih rešitev $N(S_M) = \{S_1, S_2, \ldots, S_k\}$. Sosednje rešitve se razlikujejo od rešitve S_M v tem, da imajo različne vrednosti komponent rešitve (npr. odstranjeno ali dodano kakšno vozlišče, če rešujemo problem na grafu).

Glede na to, da imajo različni kombinatorični problemi različne omejitve, moramo ločiti med **sprejemljivimi**, angl. *feasible*, in **nesprejemljivimi**, angl. *infeasible*, rešitvami [5, 10]. Sprejemljiva rešitev izpolnjuje vse omejitve, nesprejemljiva rešitev pa krši vsaj eno izmed omejitev določenega kombinatoričnega problema. Pri določenih problemih, npr. problemu trgovskega potnika, so vse možne rešitve tudi sprejemljive (če kot vse možne rešitve razumemo vse možne Hamiltonove cikle na določenem grafu), v splošnem pa to ni nujno. Pri pripravi metahevristik za reševanje problemov, katerih prostor vseh možnih rešitev vsebuje tudi nesprejemljive rešitve, se pogosto pojavi dilema, ali naj metahevristika deluje tudi nad nesprejemljivimi rešitvami ali naj

Tabela 3.1: Oznake vozlišč v grafu

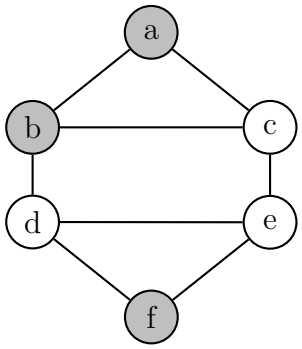
	Vključeno	**Izključeno**
Ne krši omejitev	Sprejemljivo	Prosto
Krši omejitve	Nesprejemljivo	Zasedeno

se jim izogne. V prvem primeru je potrebno poskrbeti za ustrezno spremembo (popravilo) nesprejemljivih rešitev v sprejemljive rešitve.

Kadar obravnavamo optimizacijske probleme na grafu in obstaja možnost nesprejemljivih rešitev, vsakemu vozlišču dodelimo oznako. Kot vidimo v tabeli 3.1, vozliščem, ki ne kršijo omejitev problema, pravimo **sprejemljiva**, kadar so vključena v rešitev, in **prosta**, kadar niso vključena v rešitev. Analogno vozliščem, ki kršijo omejitve problema, pravimo **nesprejemljiva**, kadar so vključena v rešitev, in **zasedena**, kadar niso vključena v rešitev.

Na sliki 3.1 vidimo problem neodvisne množice, v katerem vozlišči a in b kršita omejitev, da med vozlišči, vključenimi v rešitev, ne sme biti povezav. Zato ju označimo kot nesprejemljivi vozlišči. Za izključena vozlišča c, d in e rečemo, da so zasedena, saj je vsako izmed njih sosednje vsaj enemu izmed že vključenih vozlišč. Vidimo lahko tudi, da vozlišče f ne krši omejitev problema, saj nobeno izmed njemu sosednjih vozlišč ni vključeno v rešitev, zato ga označimo kot sprejemljivo vozlišče.

Slika 3.1: Primer kršitve omejitev problema neodvisne množice

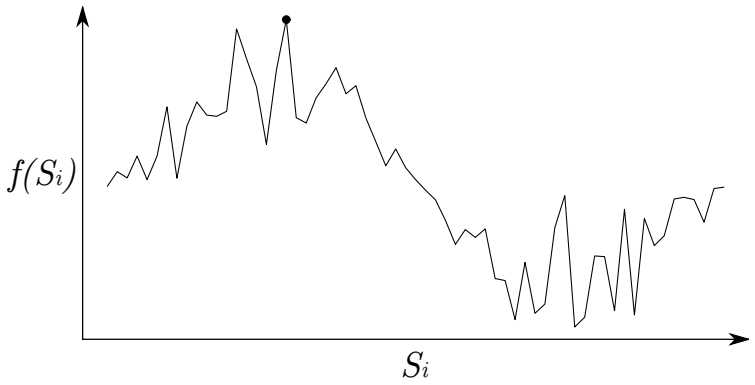


Za metahevristike sta med drugim pomembna pojma **lokalni optimum**, angl. *local optimum*, in **globalni optimum**, angl. *global optimum*, ki ju pripišemo posameznim rešitvam. Za rešitev S_L, ki je lokalni optimum, velja, da nobena izmed sosednjih rešitev iz množice $N(S_L)$ ni boljša od S_L.

Brez izgube na splošnosti se omejimo na maksimizacijske probleme in zapišemo $\forall S_i \in N(S_L) : f(S_L) \geq f(S_i)$. Za globalni optimum S_G velja, da nobena druga rešitev izmed vseh možnih ni boljša od S_G. Podobno za lokalni optimum zapišemo $\forall S_i \in S : f(S_G) \geq f(S)$, kjer je S množica vseh možnih rešitev.

Primer problema lokalnega optimuma je prikazan na sliki 3.2. S polnim krogcem je označen globalni maksimum. Vsak izmed preostalih vrhov, ki so nižji od največjega vrha, predstavlja lokalni maksimum. Iz tega neposredno sledi, da globalnega optimuma ni mogoče najti s preprosto (in naivno) metodo pomikanja v smeri izboljševanja kakovosti rešitve, dokler je to mogoče. V angleščini se takšna metoda imenuje *hill climbing* in je podrobno opisana v razdelku 3.2.1.

Slika 3.2: Primer problema lokalnega optimuma



Na področju metahevristik sta se v zadnjem času uveljavila tudi pojma **razpršitev**, angl. *diversification*, in **izostritev**, angl. *intensification* [5]. Pri razpršitvi želimo razširiti področje preiskovanja prostora rešitev z namenom prehoda v področje, ki vsebuje boljše rešitve. Izostritev, ki je nasprotna razpršitvi, govori o temeljitem preiskovanju določenega področja, z namenom da se najde čim boljša rešitev znotraj tega področja. Kot primer razpršitve lahko navedemo zamenjavo soseščine z namenom preiskovanja dela prostora rešitev, ki prej še ni bil preiskan. Izostritev lahko ponazorimo z lokalnim iskanjem znotraj določenega področja, pri čemer nas zanimajo najboljše kakovosti rešitev znotraj tega področja.

3.1.1 Lokalno iskanje

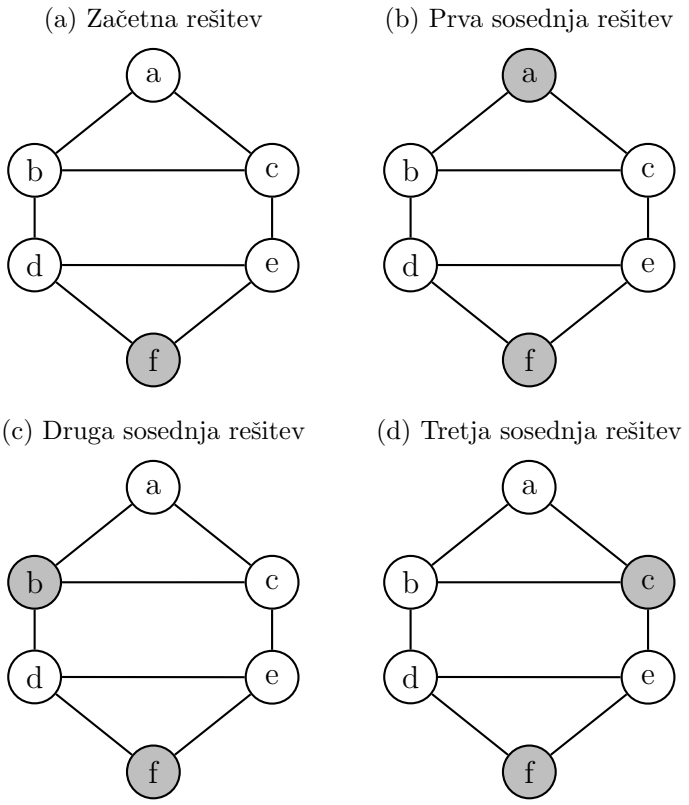
Lokalno iskanje je skupno ime za metahevristične algoritme, ki uporabljajo idejo soseščine. Vhod v lokalno iskanje je navadno začetna rešitev. Načinov,

kako generirati začetno rešitev je več. V praksi se pogosto uporabljajo konstrukcijske hevristike ali preprosto naključno generiranje rešitev.

Med izvajanjem metahevrističnega algoritma se na vsakem koraku zamenja trenutna rešitev z eno izmed sosednjih rešitev. Postopek se ponavlja, dokler ni izpolnjen ustavitveni pogoj (npr. presežen čas izvajanja, dosežena kakovost rešitve ali zaznana konvergenca).

Metodo lokalnega iskanja bomo ponazorili na problemu največje neodvisne množice. Za rešitev S definiramo okolico rešitve $N(S)$ kot množico rešitev, ki imajo dodano eno izmed prostih vozlišč. Primer je ilustriran na sliki 3.3. Začetna rešitev je prikazana na podsliki (a), na preostalih podslikah (b), (c) in (d) so prikazane vse možne sosednje rešitve.

Slika 3.3: Primer lokalnega iskanja na problemu največje neodvisne množice



3.2 Primeri metahevristik

V literaturi srečamo veliko število metahevrističnih algoritmov. Razpon tako njihove splošnosti kot tudi uspešnosti je velik. V večini primerov si „splošnost“ in „uspešnost“ nasprotujeta ter je zaradi tega potrebno iskati kompromise. Z izrazom splošnost mislimo na število različnih problemov, ki jih je mogoče reševati brez zahtevnih prilagoditev z določenim metahevrističnim algoritmom. Z izrazom uspešnost mislimo na kakovost rešitev, povprečeno čez raznolike primerke določenega problema.

Metahevristične algoritme ločimo po različnih kriterijih, med drugim glede na temelj algoritma. Določeni algoritmi temeljijo na idejah iz narave (npr. simulirano ohlajanje, sistem mravelj, evolucijski algoritem), medtem ko imajo drugi temelj v statistiki (npr. metoda navzkrižne entropije), spet tretji nimajo posebnih temeljev (npr. iskanje s tabujem, angl. *tabu search*). V nadaljevanju si bomo pogledali šest metahevrističnih algoritmov, pripravljenih za reševanje problema največje neodvisne množice. Vsi, razen evolucijskega algoritma in metode navzkrižne entropije, temeljijo na metodi lokalnega iskanja z različnimi dodatki.

3.2.1 Požrešno iskanje

Angl. *Hill climbing*. Algoritem za požrešno iskanje (glej algoritem 3.1) je eden izmed najbolj preprostih algoritmov lokalnega iskanja. Kot že samo ime pove, je algoritem požrešen, in to pomeni, da se vedno premika v smeri izboljšujoče ali enakovredne kakovosti rešitve. Ta požrešnost ni brez cene, saj tak algoritem običajno zelo hitro doseže lokalni optimum in zaključi izvajanje, ker nima mehanizma, s katerim bi se rešil iz lokalnega optimuma. Posledično je kakovost rešitve, dosežena s tem algoritmom, v splošnem nizka.

Funkciji

- GenerirajZačetnoRešitev(): generira in vrne začetno rešitev (npr. s konstrukcijsko metodo).
- SosednjeRešitve(rešitev): vrne množico vseh sosednjih rešitev, glede na defnicijo sosednosti.

3.2.2 Simulirano ohlajanje

Angl. *Simulated annealing*. Algoritem simuliranega ohlajanja (glej algoritem 3.2), na kratko „simulirano ohlajanje“, je bil prvič predstavljen leta 1983 v

Algoritem 3.1 Požrešno iskanje

```
1:  function PožrešnoIskanje()
2:      rešitev ← GenerirajZačetnoRešitev()
3:      premikUspel ← True
4:      while premikUspel do
5:          sosednjeRešitve ← SosednjeRešitve(rešitev)
6:          premikUspel ← False
7:          rešitevNova ← rešitev
8:          for all rešitev' ∈ sosednjeRešitve do
9:              if f(rešitev') ≥ f(rešitevNova) then
10:                 rešitevNova ← rešitev'
11:                 premikUspel ← True
12:         if premikUspel then
13:             rešitev ← rešitevNova
14:     return rešitev
```

[11]. Opis, izboljšave in variacije simuliranega ohlajanja je moč najti v [12, 13]. Opis uporabe algoritma simuliranega ohlajanja za reševanje problema največjega polnega podgrafa najdemo v [14] in za reševanje njemu ekvivalentnega problema, problem najmanjšega pokritja grafa, najdemo v [15].

Glavna ideja algoritma simulirano ohlajanje je v tem, da se izvede veliko število iteracij in v vsaki izmed njih se generira sosednja rešitev trenutne rešitve. Sosednja rešitev, ki je boljša ali vsaj enako dobra, vedno nadomesti trenutno rešitev. Slabša sosednja rešitev nadomesti trenutno rešitev samo z določeno verjetnostjo, ki je odvisna od temperature. Temperatura se skozi iteracije znižuje glede na **urnik ohlajanja**. Nižja, kot je temperatura, manjša je verjetnost, da slabša sosednja rešitev nadomesti trenutno rešitev.

Urnik ohlajanja je statičen ali dinamičen. Pri statični verziji je število iteracij pri konstantni temperaturi določeno vnaprej, s parametri. Po drugi strani je pri dinamični verziji število iteracij pri določeni temperaturi odvisno od tega, kako se rešitev izboljšuje (še vedno obstaja možnost zgornje meje, preko parametra). V algoritmu 3.2 vidimo različico, ki uporablja statičen urnik ohlajanja [13].

Generiranje rešitev na začetku lahko naredimo na različne načine. Eden izmed njih je, da začnemo s prazno rešitvijo in dodajamo eno po eno prosto vozlišče, dokler je to mogoče.

Pri določanju sosednjih rešitev je na voljo veliko različic in ena izmed njih je sledeča. Iz veljavne rešitve najprej naključno odstranimo določeno število

vključenih vozlišč k_{odst}, linearno odvisno od temperature c, $k_{odst} = \frac{c}{10}$. Nato v naključnem vrstnem redu vključujemo prosta vozlišča, dokler je to mogoče.

Parametri

- α: parameter statičnega urnika ohlajanja.
- $c_{zač}$: določa začetno temperaturo.
- c_{kon}: določa končno temperaturo.
- L: določa število iteracij pri konstantni temperaturi.

Algoritem 3.2 Simulirano ohlajanje

```
1:  function SimuliranoOhlajanje(α, c_zač, c_kon, L)
2:      rešitev ← GenerirajZačetnoRešitev()
3:      c ← c_zač
4:      while c ≥ c_kon do
5:          for i = 1 to L do
6:              soseda ← Soseda(rešitev)          // Pridobi naključno sosedo
7:              if f(soseda) ≥ f(rešitev) then
8:                  rešitev ← soseda              // Vedno sprejmi boljšo ali enako
9:              else if exp((f(soseda)−f(rešitev))/c) ≤ rand() then
10:                 rešitev ← soseda
11:         c ← c · α                             // Zmanjšaj temperaturo
12:     return rešitev
```

Funkcije

- GenerirajZačetnoRešitev(): generira in vrne začetno naključno rešitev.
- Rand(): vrne naključno vrednost iz intervala $[0, 1)$.
- Soseda(rešitev): vrne naključno izbrano sosednjo rešitev.

3.2.3 Razpršeno iskanje

Angl. *Scatter search*. Uvod v algoritem razpršenega iskanja (glej algoritem 3.3), na kratko „razpršeno iskanje“, je podan v [16], več podrobnosti najdemo v [17]. Razpršeno iskanje je metoda, ki temelji na težnji po čim večji preiskanosti prostora rešitev v smislu, da se pregleda čim večje število raznolikih rešitev.

Razpršeno iskanje je bilo že uporabljeno za reševanje problema največjega polnega podgrafa [18]. Na začetku razpršenega iskanja se generira referenčna množica, ki je glavna sestavina tega algoritma, tako da vsebuje p raznolikih rešitev. Kasneje se v vsaki iteraciji posodobi, tako da vsakič vsebuje b_1 najboljših rešitev in b_2 najbolj raznolikih rešitev (v primerjavi s tistimi rešitvami, ki so že v referenčni množici). V vsaki iteraciji so pari rešitev med seboj kombinirani. Eden izmed možnih načinov je, da nad vsemi možnimi pari rešitev izvedemo naključno križanje, angl. *random uniform two-parent crossover*. Ker pri tem nastanejo tudi nesprejemljive rešitve, moramo uporabiti metodo za popravilo rešitev, ki ji sledi metoda za izboljšavo rešitev.

Pomemben princip razpršenega iskanja je ta, da se v referenčni množici ne smejo pojavljati podvojene rešitve. To se lahko preveri ali ob samem vstavljanju v referenčno množico ali pa se naknadno izvede preveritev in izločijo podvojene rešitve.

Opis razpršenega iskanja v [16] določa uporabo konvergenčnega kriterija, ki je izpolnjen, ko se referenčna množica med dvema iteracijama ne spremeni več. Ker je mogoče, da to traja zelo dolgo, lahko uporabimo tudi drugačen konvergenčni kriterij, pri katerem se algoritem zaključi, ko je prišlo do določenega števila zaporednih iteracij, v katerih se kakovost rešitve ni izboljšala.

Parametri

- p: število naključnih rešitev, ki naj se generirajo na začetku za zalogo.
- b_1: število najboljših rešitev, ki naj bodo del referenčne množice.
- b_2: število najbolj različnih rešitev, ki naj bodo del referenčne množice.
- maksIterBrezIzboljšave: ustavitveni pogoj, največje število zaporednih iteracij, v katerih se kakovost rešitve ni izboljšala.

Funkcije

- GenerirajPare(refMnožica): vrne vse možne pare kombinacij rešitev iz refMnožica.
- GenerirajRešitve(p): vrne množico p naključno generiranih rešitev. Rešitve so generirane naključno, začenši s prazno rešitvijo in z dodajanjem enega po enega prostega vozlišča, dokler je to mogoče.
- IzboljšajRešitve(zaloga): izvede izboljšavo rešitev. Pri tem se eno po eno vključujejo prosta vozlišča, dokler je to mogoče.

Algoritem 3.3 Razpršeno iskanje

```
1: function RazpršenoIskanje(p, b_1, b_2)
2:     zaloga ← GenerirajRešitve(p)                    // Začetne rešitve
3:     refMnožica ← NajboljšaKakovost(zaloga, b_1) ∪
           NajboljšaRazličnost(zaloga, refMnožica, b_2)
4:     iterBrezIzboljšave ← 0                          // Števec za ustavitveni pogoj
5:     najboljšaRešitev ← Najboljša(refMnožica)
6:     while iterBrezIzboljšave < maksIterBrezIzboljšave do
7:         zaloga ← {}
8:         pari ← GenerirajPare(refMnožica)
9:         for all par ∈ pari do
10:            NarediKombinacijo(par) // Naključno enakomerno križanje
11:            zaloga ← zaloga ∪ par
12:        PopraviRešitve(zaloga)
13:        IzboljšajRešitve(zaloga)
14:        IzločiPodvojene(zaloga, refMnožica)
15:        zaloga ← zaloga ∪ refMnožica
16:        refMnožica ← NajboljšaKakovost(zaloga, b_1) ∪
               NajboljšaRazličnost(zaloga, refMnožica, b_2)
17:        if f(Najboljša(refMnožica)) ≥ f(najboljšaRešitev) then
18:            najboljšaRešitev ← Najboljša(refMnožica)
19:            iterBrezIzboljšave ← 0
20:        else
21:            iterBrezIzboljšave++          // Brez izboljšave najboljše rešitve
22:    return Najboljša(refMnožica)
```

- IzločiPodvojene(zaloga, refMnožica): iz zaloge izloči vse takšne rešitve, ki so že prisotne v referenčni množici.
- Najboljša(refMnožica): vrne najboljšo rešitev glede na kakovost.
- NajboljšaKakovost(zaloga, k): vrne k najboljših rešitev iz zaloge in jih pri tem izloči iz zaloge.
- NajboljšaRazličnost(zaloga, refMnožica, k): vrne k najbolj različnih rešitev iz zaloge v primerjavi z rešitvami v refMnožica in jih pri tem izloči iz zaloge. Različnost se meri kot evklidska razdalja med rešitvami čez vse kombinacije rešitev v zalogi in refMnožica. Najbolj različne se poiščejo s pomočjo max-min kriterija, kot je opisano v razdelku 2 v [16].
- NarediKombinacijo(par): izvede kombinacijo med dvema rešitvama z uporabo enakomernega naključnega križanja, angl. *uniform random crossover*, vozlišče po vozlišče.
- PopraviRešitve(zaloga): za vsako rešitev v zalogi izvede popravilo vseh nesprejemljivih rešitev. Pri tem izloča eno po eno nesprejemljivo vozlišče, v naključnem vrstnem redu, dokler rešitev ne postane sprejemljiva.

3.2.4 Metoda navzkrižne entropije

Angl. *Cross-entropy method*. Metoda navzkrižne entropije (glej algoritem 3.4), na kratko „navzkrižna entropija“, je bila predstavljena v [19]. Med drugim je namenjena tudi reševanju problemov kombinatorične optimizacije. V jedru navzkrižne entropije sta dva koraka, ki se običajno večkrat zaporedno ponavljata. V prvem koraku se izvede generiranje naključnih vzorcev z določenim mehanizmom in v drugem koraku posodabljanje parametrov mehanizma, ki izvaja generiranje naključnih vzorcev, s ciljem, da bodo imeli v naslednji generaciji naključni vzorci boljšo kakovost.

Navzkrižno entropijo za reševanje problema največjega polnega podgrafa so že uporabili v [20, 21]. Oboji so kot mehanizem za generiranje naključnih vzorcev vzeli matriko verjetnosti V, ki določa verjetnost za vključitev vozlišča v_j takoj zatem, ko je bilo vključeno vozlišče v_i, označimo kot „prehod“. Matrika je velikosti $(n+1) \times n$, pri čemer je n število vozlišč grafa, in ima eno vrstico več, kot je vozlišč, ker prva vrstica matrike določa, kakšne so verjetnosti, da se začne graditi rešitev s posameznim vozliščem. Preostale vrstice določajo verjetnost, da se za vozliščem v_i vključi vozlišče v_j. Vozlišče v_j je za vozliščem v_i vključeno v rešitev z verjetnostjo $V[i+1, j]$. Pri tem uporabimo Bernoullijevo porazdelitev z začetno verjetnostjo $\frac{1}{n}$ za vsak prehod.

Kljub temu da v [20, 21] problem generiranja nesprejemljivih rešitev ni neposredno omenjen, do njega pride in ga je potrebno rešiti. Eden izmed načinov, kako ga rešimo, je, da postavimo verjetnosti vseh zasedenih vozlišč na nič. Katera vozlišča so zasedena, je potrebno preračunati po vsaki vključitvi nekega vozlišča. Poleg tega moramo na nič postaviti tudi verjetnosti za vključitve vozlišč, ki so že vključena v rešitev.

Navzkrižno entropijo je mogoče kombinirati z lokalnim iskanjem [21], vendar to pripelje do izgube izraza navzkrižne entropije, saj ni več nujno, da je navzkrižna entropija tista, ki je odgovorna za generiranje dobrih rešitev. Zato bi morali preveriti, kako se metoda obnaša brez uporabe lokalnega iskanja in z uporabo lokalnega iskanja, in rezultate primerjati. V nadaljevanju bomo obravnavali navzkrižno entropijo, ki ne uporablja lokalnega iskanja.

Parametri

- ρ: določa, kakšen delež vzorca je uporabljen pri posodobitvi parametrov.
- c: določa velikost vzorca v kombinaciji z določenim parametrom primerka (število vozlišč).
- α: določa količino glajenja parametrov, angl. *smoothing*.

Funkcije

- Dodaj(urejeniSeznam, rešitev): doda rešitev v urejeni seznam glede na kakovost.
- GenerirajRešitev(V): vrne novo rešitev, ki je generirana s pomočjo Bernoullijeve porazdelitve z uporabo verjetnosti, ki so podane v matriki V.
- PosodobiVerjetnosti(V, α, urejeniSeznam): posodobi matriko verjetnosti V glede na rešitve v urejenem seznamu. Parameter α določa glajenje, tako da je α-delež nove verjetnosti združen z $(1 - \alpha)$-deležem stare verjetnosti za določen prehod.
- Prvi(urejeniSeznam): vrne rešitev, ki je na začetku urejenega seznama (tj. najboljša rešitev).
- Skrajšaj(urejeniSeznam, N_{najb}): v urejenem seznamu zadrži samo N_{najb} najboljših rešitev, ostale zanemari.

Algoritem 3.4 Navzkrižna entropija

```
1:  function NavzkrižnaEntropija(ρ, c, α)
2:      V ← {{1/n, ..., 1/n}, ..., {1/n, ..., 1/n}}          // Matrika verjetnosti
3:      N ← n × c                                              // Velikost vzorca
4:      N_najb ← N × ρ                  // Število rešitev vzorca za posodobitev
5:      urejeniSeznam ← {}                  // Rešitve razvrščene po kakovosti
6:      iterBrezIzboljšave ← 0
7:      najboljšaRešitev ← 0
8:      while iterBrezIzboljšave < maksIterBrezIzboljšave do
9:          for i = 1 to N do                          // Generiraj vzorčne rešitve
10:             rešitev ← GenerirajRešitev(V)
11:             Dodaj(urejeniSeznam, rešitev)          // Dodaj rešitev v seznam
12:         Skrajšaj(urejeniSeznam, N_najb)    // Uporabi samo N_najb rešitev
13:         PosodobiVerjetnosti(V, α, urejeniSeznam)
14:         if f(Prva(urejeniSeznam)) ≥ f(najboljšaRešitev) then
15:             najboljšaRešitev ← Prva(urejeniSeznam)
16:             iterBrezIzboljšave ← 0
17:         else
18:             iterBrezIzboljšave++
19:     return Prva(urejeniSeznam)
```

3.2.5 Evolucijski algoritem

Angl. *Evolutonary algorithm*. Evolucijski algoritem (glej algoritem 3.5) je opisan v [22, 23]. Poleg tega so določene zanimive ideje glede reševanja problema nesprejemljivih rešitev po mutaciji in/ali križanju podane v [24]. Evolucijski algoritem je osnovan na procesu evolucije iz biologije in pri izvajanju dela z določeno populacijo, katere člani so mutirani in med seboj križani. Mutacije in križanje je mogoče izvesti na veliko načinov. Za mutacije je najpreprostejši primer ta, da se vsaka komponenta rešitve spremeni z določeno verjetnostjo (npr. na grafu vključitev določenega vozlišča, ki je bilo prej izključeno, ali obratno). Pri križanju obravnavamo par rešitev in z določeno verjetnostjo paroma zamenjamo določeno komponento rešitve. Mutacijam in križanju sledi postopek selekcije. Selekcija običajno vključuje kriterij kakovosti rešitve, lahko pa tudi druge kriterije, npr. raznolikost rešitev, da se prepreči prehitra konvergenca.

Parametri

- λ: velikost populacije.
- μ: število potomcev, ki naj se jih naredi pri križanju (na iteracijo).
- n_m: število mutacij, ki naj se jih izvede na vsaki rešitvi.

Algoritem 3.5 Evolucijski algoritem

```
1:  function EvolucijskiAlgoritem(λ, μ, n_m)
2:      populacija ← GenerirajZačetnoPopulacijo(λ)
3:      ustavitveniPogoj ← False
4:      while ¬ ustavitveniPogoj do
5:          populacijaMutirano ← IzvediMutacije(populacija, n_m)
6:          križanci ← IzvediKrižanje(populacija)
7:          križanciMutirano ← IzvediMutacije(križanci, n_m)
8:          populacija ← NovaGeneracija(populacija ∪ križanci ∪
                populacijaMutirano ∪ križanciMutirano)
9:          ustavitveniPogoj ← PreveriUstavitveniPogoj()
10:     return NajboljšaRešitev(populacija)
```

Funkcije

- GenerirajZačetnoPopulacijo(λ): pripravi začetno populacijo velikosti λ.

- IzvediKrižanje(populacija): izvede parno križanje med rešitvami v populaciji, pri čemer sta dva starša nadomeščena z dvema potomcema.
- IzvediMutacije(populacija, n_m): vrne novo množico rešitev, pri čemer je bilo nad vsako rešitvijo izvedenih n_m mutacij (naključno porazdeljene).
- PreveriUstavitveniPogoj(): ugotovi, ali je ustavitveni pogoj izpolnjen.
- NajboljšaRešitev(populacija): vrne najboljšo rešitev v populaciji glede na kakovost.
- NovaGeneracija(populacija): na podlagi podane populacije vrne novo populacijo glede na določene kriterije (npr. kakovost, raznolikost rešitev).

3.2.6 Sistem mravelj

Sistemov mravelj je več, dva izmed bolj znanih sta angl. *Ant system* in angl. *Ant colony system*. Ena izmed različic Ant system se imenuje $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System in je opisana v [25]. Opažanje, da ima $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System podobno zmogljivost kot ACS, najdemo v [25]. Poleg tega trdijo, da so se vse ostale preizkušene implementacije sistema mravelj odrezale slabše. V [26] je podan jedrnat pregled različnih implementacij idej sistema mravelj. Med drugim je opisan Ant colony system (ACS) in podani so začetni poskusi analize časovne zahtevnosti opisanih algoritmov.

Sistem mravelj temelji na grafu in je zato najbolj primeren za reševanje NP-polnih in NP-ekvivalentnih problemov na grafih (npr. problem trgovskega potnika). Glavna ideja je v tem, da je na voljo določeno število mravelj, ki se jih odloži v graf. Njihova naloga je, da obiščejo različna vozlišča. V vsakem vozlišču se morajo odločiti, kako nadaljevati. To določajo vrednosti fermonov, ki so pripisane vsakemu izmed vozlišč ali vsaki izmed povezav grafa. Vrednosti fermonov na vozliščih ali povezavah je treba obravnavati kot uteži verjetnosti, da mravlja gre po določeni poti. Večja kot je vrednost fermona, večja je verjetnost, da gre mravlja po tej poti. Pri tem je potrebno upoštevati morebitne prepovedi določenih povezav (npr. če je bilo neko vozlišče že obiskano, pri problemu trgovskega potnika). Če vrednosti fermonov pripišemo povezavam, lahko v splošnem zapišemo verjetnost, da se mravlja premakne iz vozlišča v_i v vozlišče v_j, glej enačbo 3.1. S $\tau_{i,j}$ smo označili vrednost fermona na povezavi med vozliščema v_i in v_j in z vsoto ponazorili seštevek vrednosti fermonov vseh povezav, ki izhajajo iz vozlišča v_i.

$$p_{i,j} = \frac{\tau_{i,j}}{\sum_{k=1}^{n} \tau_{i,k}} \tag{3.1}$$

Za določene vrste problemov (npr. problem največje neodvisne množice) vrsta grafa ni enolično določena. Pri sami implementaciji izbiramo na primer med polnim grafom, verižnim grafom in še nekaterimi drugimi vrstami grafov [26]. Od vrste grafa je odvisna hitrost in uspešnost reševanja problema. V algoritmu 3.6 si oglejmo eno od možnih implementacij Ant System na verižnem grafu.

Parametri

- M: število mravelj v sistemu.
- τ_{max}: največja dovoljena vrednost fermona.
- ρ: določa jakost izhlapevanja fermonov.
- razmerjePosodobitev: določa razmerje med uporabo najboljše rešitve v iteraciji ali najboljše rešitve do sedaj (kadar je enak 1, se uporablja samo najboljša rešitev v iteraciji, in kadar je enak 0, se uporablja samo najboljša rešitev do sedaj).

Funkcije

- IzhlapevanjeFermonov(τ, ρ): izvede izhlapevanje fermonov v matriki τ, pri čemer so vse vrednosti pomnožene z $(1 - \rho)$. Poleg tega se upoštevajo tudi zgornje in spodnje meje za vrednosti fermonov.
- OdložiFermone(τ, rešitev): odloži fermone na pot, ki jo je opravila določena mravlja. Ko se določa nova vrednost fermonov, se ne prekorači zgornje meje.
- PreveriUstavitveniPogoj(): ugotovi, ali je ustavitveni pogoj izpolnjen.
- VključiVozlišče(rešitev, v_k): če je vozlišče v_k prosto, ga vključi v rešitev.

Algoritem 3.6 Sistem mravelj

```
1: function SistemMravelj(M, ρ, τ_max, razmerjePosodobitev)
2:     for i = 1 to n do
3:         τ[i][0] ← τ_max   // Inicializacija vseh fermonov na zgornjo vrednost
4:         τ[i][1] ← τ_max
5:     ustavitveniPogoj ← False
6:     while ¬ ustavitveniPogoj do
7:         for mravljaŠt = 1 to M do
8:             k ← Rand()                          // Naključno začetno vozlišče
9:             rešitev ← {v_k}
10:            for i = 1 to n do
11:                k ← (k + 1) mod n
12:                if τ[k][0]/(τ[k][0] + τ[k][1]) < Rand() then
13:                    VključiVozlišče(rešitev, v_k)
14:            if f(rešitev) > f(najboljšaVIteraciji) then
15:                najboljšaVIteraciji ← rešitev
16:        if f(najboljšaVIteraciji) > f(najboljšaDoSedaj) then
17:            najboljšaDoSedaj ← najboljšaVIteraciji       // Nova najboljša
18:            τ_max = f(najboljšaVIteraciji)/ρ
19:            τ_min = τ_max/a
20:        IzhlapevanjeFermonov(τ, ρ)
21:        if razmerjePosodobitev < Rand() then
22:            τ ← OdložiFermone(τ, najboljšaVIteraciji)
23:        else
24:            τ ← OdložiFermone(τ, najboljšaDoSedaj)
25:        ustavitveniPogoj ← PreveriUstavitveniPogoj()
26:    return najboljšaDoSedaj
```

Poglavje 4

Vzporedni stroj

V preteklih desetletjih so bili za hitrejšo obdelavo večje količine podatkov (reševanje kompleksnih problemov ipd.) razviti večprocesorski računalniki in vzporedni stroji, pri slednjih se med seboj povezuje večje število računalnikov. V zadnjih letih je mogoče opaziti, da se je poudarek preselil na razvoj večjedrnih procesorjev. To se je v večji meri zgodilo zaradi dosega fizičnih omejitev pri povečevanju frekvenc procesorjev [27]. Uveljavitev večjedrnih procesorjev je povečala možnosti za vzporedno izvajanje algoritmov, in to pri nižjih stroških kot nekdaj. Poleg tega je napredek pri povezovanju računalnikov med seboj preko hitrega omrežja (npr. Ethernet, InfiniBand) omogočil sestavo učinkovitih in ne preveč dragih vzporednih strojev.

Napredek strojne opreme je gnal razvoj različnih orodij za izkoriščanje potenciala vzporednih strojev. S pojmom **zrnatost** označujemo, na kakšnem nivoju med seboj komunicirajo algoritmi. Kolikor je izmenjava podatkov pogosta in se procesi v večji meri zanašajo drug na drugega, to imenujemo **fina zrnatost**. V nasprotnem primeru, ko si procesi podatke izmenjujejo bolj poredko in so pri izvajanju bolj samostojni, to imenujemo **groba zrnatost**.

Glavni predstavniki fine zrnatosti so vmesnik MPI [28], vmesnik OpenMP [4] in PVM [29]. Implementacij vmesnika MPI je večje število (FT-MPI, HP MPI, LA-MPI, LAM/MPI, MPICH, OpenMPI, PACX-MPI itd.).

Pri grobi zrnatosti izpostavimo računalništvo na mreži [30], angl. *grid computing*, in računalništvo v oblaku [31], angl. *cloud computing*. Obstaja več implementacij računalništva na mreži, med njimi je ena od bolj znanih Globus Toolkit [32].

Eden izmed boljših virov informacij o pripravi in uporabi vzporednih algoritmov fine zrnatosti je [4], kjer je opis mnogih tehnik priprave vzporednih algoritmov, teorije o analizi učinkovitosti in pospešitvi vzporednih algoritmov,

kako najti in izkoristiti vire vzporednosti, kakšni so vzorci komunikacije in kako se prilegajo različnim omrežjem, opis MPI in OpenMP ter opis in analiza množice vzporednih algoritmov za reševanje različnih problemov (sortiranje, množenje matrik, problemi na grafih, FFT itd.).

Po [4] obstajajo različne tehnike dekompozicije, in sicer rekurzivna dekompozicija (uporablja princip „deli in vladaj“), podatkovna (uporablja delitev podatkov na manjše kose), raziskovalna dekompozicija (uporablja razdelitev prostora rešitev) in napovedovalna dekompozicija (princip podoben napovedovanju skokov na procesorju). Prvi dve tehniki, rekurzivna in podatkovna dekompozicija, sta splošni, saj ju je mogoče uporabiti na veliki množici različnih problemov. Preostali tehniki sta specializirani in zato uporabni le na določenih problemih. Seveda je mogoča tudi poljubna kombinacija zgoraj naštetih tehnik, odvisno od potreb in lastnosti problema.

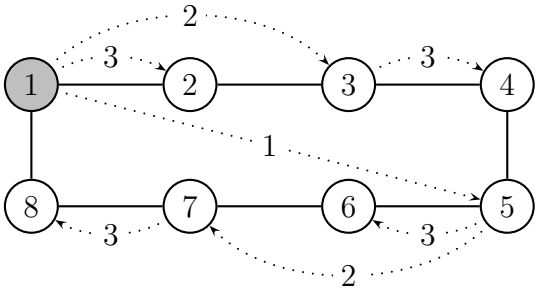
V nadaljevanju bomo za lažjo razlago predpostavili, da se vsak proces izvaja na svojem sistemu, označili ga bomo kot vozlišče. Vozlišča so med seboj povezana v omrežje in tvorijo topologijo. Naštejmo nekaj bolj pogostih topologij: 2-dimenzionalna mreža, obroč, polno povezano omrežje, skupno vodilo, zaporedje, zvezda itd. [4]. Od tega, kakšno topologijo imamo na voljo, je odvisno, kako učinkovit bo prenos podatkov.

Ker je hitrost prenosa določene količine podatkov preko omrežja bistveno manjša od hitrosti izvajanja programov na procesorju, je zelo pomembno, na kakšen način se preko omrežja prenašajo podatki. Ta način prenosa mora biti prilagojen topologiji omrežja. Kot primer prenašanja podatkov med procesi si oglejmo razpošiljanje, angl. *broadcast*, ki se pogosto pojavlja pri vzporednih algoritmih. Pri tovrstnem pošiljanju eden izmed procesov pošlje določene podatke vsem ostalim procesom, imenujemo jih **sporočilo**, angl. *message*. Na slikah 4.1, 4.2 in 4.3 smo s primeri ponazorili optimalno razpošiljanje na obroču, na 2-dimenzionalni mreži in na skupnem vodilu. S polnimi črtami so označene povezave med vozlišči in s pikčastimi črtami prenosi med vozlišči, pri čemer je na pikčasti črti zapisana tudi številka koraka. Pri vseh topologijah, razen pri skupnem vodilu, predpostavimo, da prenos sporočila med poljubnima vozliščema traja en korak in da lahko v istem koraku vsako izmed vozlišč komunicira le z enim drugim vozliščem.

Na sliki 4.1 vidimo primer razpošiljanja na obroču z osmimi procesi. Kot vidimo, se v prvem koraku kot izvorni proces obnaša samo dejanski izvorni proces. V drugem koraku se kot izvorni proces, poleg dejanskega izvornega procesa, obnaša tudi proces, ki je v prvem koraku prejel sporočilo od izvornega procesa. Takšno rekurzivno razpolavljanje se nadaljuje v tretjem koraku (in nadaljnjih korakih, če bi imeli več procesov). Zato je za razpošiljanje potrebnih

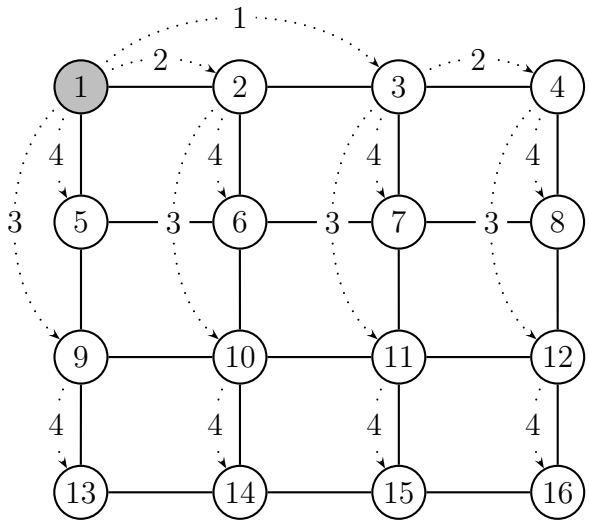
$\log_2 n$ korakov [4], torej v našem primeru trije koraki.

Slika 4.1: Primer razpošiljanja na obroču ($p = 8$, 3 koraki)



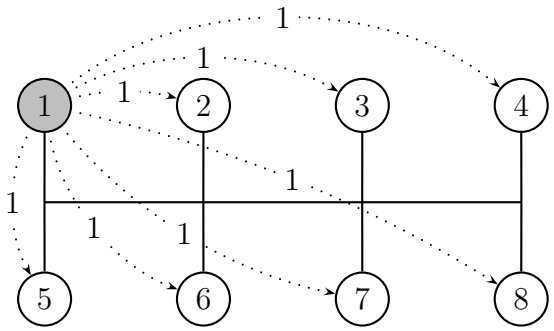
Nadalje na sliki 4.2 vidimo primer razpošiljanja na 2-dimenzionalni mreži s 16 procesi. Kot pri obroču lahko tudi na mreži uporabimo rekurzivno razpolavljanje in opravimo razpošiljanje v $\log_2 n$ korakih [4]. Zato so v primeru 16 procesov potrebni štirje koraki.

Slika 4.2: Primer razpošiljanja na 2-dimenzionalni mreži ($p = 16$, 4 koraki)



Zadnjega izmed treh primerov, razpošiljanje na skupnem vodilu, si oglejmo na sliki 4.3. Zaradi posebne lastnosti vodila, pri katerem eden izmed procesov pošilja, vsi ostali pa sprejemajo, zadošča en sam korak, ne glede na število procesov.

Slika 4.3: Primer razpošiljanja na vodilu ($p = 8$, 1 korak)



Pospešitev označimo z S_p in definiramo kot razmerje med časom izvajanja najboljšega znanega zaporednega algoritma v razmerju do časa izvajanja vzporednega algoritma, ki reši enak problem z uporabo p procesnih elementov. Pri tem se predpostavlja, da je vsak izmed p procesnih elementov enak procesnemu elementu, na katerem je bil izmerjen čas zaporednega algoritma [4]. Če s T_1 označimo čas izvajanja najboljšega zaporednega algoritma in s T_p čas izvajanja zaporednega algoritma, zapišemo pospešitev z enačbo 4.1.

$$S_p = \frac{T_1}{T_p} \tag{4.1}$$

Učinkovitost označimo z E_p in definiramo kot razmerje med pospešitvijo in številom procesnih elementov p. Na idealnem vzporednem stroju je učinkovitost enaka 1 [4]. Če se vzporedni algoritem izvaja na p procesih, zapišemo učinkovitost z enačbo 4.2.

$$E_p = \frac{S_p}{p} \tag{4.2}$$

Pri pospešitvi in učinkovitosti se moramo zavedati, da je čas izvajanja zaporednega in vzporednega algoritma odvisen tudi od same implementacije in uporabljenih orodij za pripravo vzporednih algoritmov. Poleg tega lahko pri definiciji pospešitve opazimo, da je zaradi narave metahevrističnih algoritmov težje določiti, kateri je najboljši znani algoritem za reševanje določenega problema, ali pa je to celo nemogoče.

V nadaljevanju si bomo v podpoglavju 4.1 pogledali OpenMPI, ki je ena izmed implementacij vmesnika MPI, in v podpoglavju 4.2 PVM. Med seboj ju bomo primerjali v podpoglavju 4.3.

4.1 OpenMPI

OpenMPI je implementacija programskega vmesnika Message Passing Interface (MPI), kot je predpisan s strani Message Passing Interface Forum [28]. OpenMPI je nastal kot kombinacija treh uveljavljenih implementacij MPI vmesnika, FT-MPI, LA-MPI in LAM/MPI, z dodatki PACX-MPI [33].

OpenMPI za komunikacijo uporablja TCP protokol in leno vzpostavljanje povezav. Slednje pomeni, da se povezava vzpostavi šele, ko je to potrebno, in ne na začetku izvajanja programa, ki uporablja OpenMPI. Ko je povezava enkrat vzpostavljena, se je do konca izvajanja ne zapre [33]. OpenMPI se zažene in konča skupaj s programom, kar v praksi pomeni, da je strošek režije pri kratkih programih relativno velik v primerjavi z dolgimi programi.

OpenMPI vsebuje podporo za povezave, hitrejše od običajnega LAN omrežja, kot na primer InfiniBand. Za prenos podatkov preko omrežja OpenMPI avtomatsko izbere najhitrejši način povezave, ki je na voljo [33]. Podprto je veliko število skupinskih operacij, namenjenih prenosu in izmenjavi podatkov med različnimi procesi. Podroben opis najbolj pogosto uporabljenih operacij najdemo v [4], na tem mestu le naštejmo bolj pomembne: MPI_Allgather, MPI_Allreduce, MPI_Alltoall, MPI_Barrier, MPI_Bcast, MPI_Gather, MPI_Recv, MPI_Reduce, MPI_Scatter, MPI_Send, MPI_Sendrecv itd. Med njimi je operacija razpošiljanje, MPI_Bcast, ena izmed najbolj pomembnih. V nadaljevanju bomo določene izmed naštetih operacij uporabili pri vzporednih metahevrističnih algoritmih.

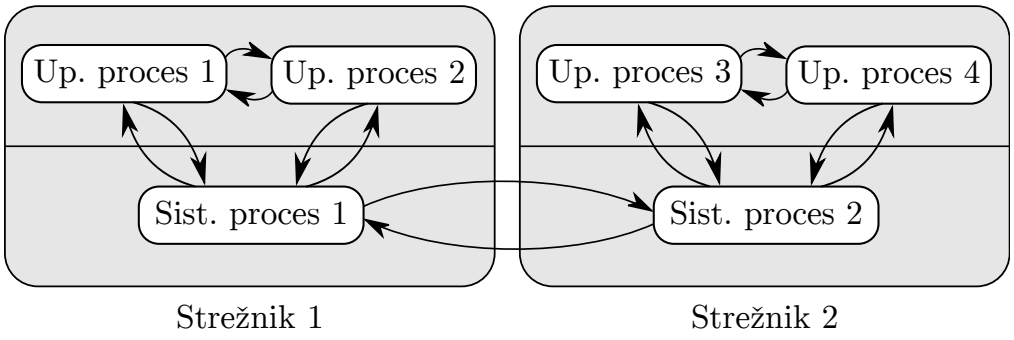
Kljub temu da po privzetih nastavitvah podpora za heterogena okolja ni vključena, jo lahko vključimo izrecno in s tem omogočimo komunikacijo med programi, ki so lahko napisani v različnih programskih jezikih: C, C++ in Fortran. Poleg tega se z vključitvijo omenjene podpore omogoči tudi izvajanje na arhitekturno heterogenih okoljih (uporaba različne strojne opreme in različnih operacijskih sistemov) [33]. Pri tem moramo izpostaviti še to, da je v splošnem hitrost komunikacije v bolj heterogenih okoljih manjša zaradi potrebnih pretvorb (npr. pretvorba iz zapisa debelega konca v zapis tankega konca).

4.2 PVM

Za razliko od OpenMPI Parallel Virtual Machine (PVM) ni implementacija vmesnika in zato obstaja le ena implementacija PVM, opisana je v [29]. PVM je navidezni vzporedni stroj, ki ga je potrebno vzpostaviti že pred začetkom izvajanja uporabniškega programa in se ne zaključi neposredno po koncu izvajanja uporabniškega programa. V PVM je mogoče dodati poljubno število he-

terogenih računalnikov (glede na strojno opremo in operacijski sistem). PVM podpira kar 38 različnih strojnih arhitektur, že od verzije 3.3 dalje (izdana leta 1994) [29]. Pri tem je pomembno opaziti, da je prenos podatkov običajno hitrejši, če je sistem bolj homogen, saj ni potrebno izvajati določenih pretvorb med različnimi sistemi. Na vsakem izmed računalnikov, vključenih v PVM, teče sistemski PVM proces, ki ima nadzor nad vsemi uporabniškimi PVM procesi. Sistemski PVM proces mora skrbeti za nemoteno komunikacijo uporabniških procesov z drugimi uporabniškimi procesi. Primer komunikacije je ponazorjen na sliki 4.4, kjer uporabniška procesa 1 in 2 komunicirata neposredno znotraj strežnika 1, enako velja tudi za uporabniška procesa 3 in 4 znotraj strežnika 2. Kadar morata med seboj komunicirati procesa, ki se nahajata na različnih strežnikih, to poteka posredno preko sistemskega PVM procesa.

Slika 4.4: Primer medstrežniške komunikacije pri PVM



Pri prenosu podatkov preko mreže PVM uporablja protokol UDP, kar pomeni, da je prenos preko omrežja dobro izkoriščen, seveda pod pogojem, da je omrežje zanesljivo in ni potrebno pogosto ponavljati pošiljanja podatkov.

Tako kot prej omenjeni OpenMPI tudi PVM podpira veliko število skupinskih operacij. Podrobnosti je mogoče najti v [29], tukaj naštejmo najpomembnejše: pvm_barrier, pvm_bcast, pvm_gather, pvm_recv, pvm_reduce, pvm_scatter, pvm_send itd. Kot pri OpenMPI bomo tudi pri PVM uporabili določene izmed naštetih operacij. Bolj natančno, če bo le mogoče, bomo uporabili ekvivalentne operacije.

4.3 Primerjava OpenMPI in PVM

Implementacije programskega vmesnika MPI se v današnjem času zelo aktivno razvijajo, medtem ko je razvoj PVM manj aktiven. Slednje vidimo v [34], saj so bile zadnje spremembe glede funkcionalnosti izdane z verzijo 3.4.4 (z dne

28.09.2001). Od takrat naprej ni bilo dodane funkcionalnosti, le odpravljanje napak in dodajanje sistemske podpore (npr. za strojno arhitekturo x86-64). Pomanjkanje razvoja se odraža tudi v dejstvu, da PVM trenutno nima podpore za hitro omrežje InfiniBand, ki je v splošni rabi na superračunalnikih.

OpenMPI za prenos podatkov preko omrežja uporablja protokol TCP, medtem ko PVM uporablja protokol UDP. Glavna razlika med protokoloma TCP in UDP je, da se pri TCP protokolu tekom prenosa podatkov vzdržuje seja, pri UDP protokolu pa sej nimamo. Poleg tega je pri TCP protokolu zagotovljeno, da bo vsak paket prispel na cilj. Pri UDP protokolu to ni zagotovljeno (za zanesljivo komunikacijo mora poskrbeti sam PVM). Glava paketa je pri TCP protokolu večja kot pri UDP protokolu, saj vsebuje več informacij.

Med OpenMPI in PVM je razlika tudi v načinu izvajanja okolja. OpenMPI se zažene skupaj s programom, medtem ko mora biti PVM pred zagonom že vzpostavljen. Zaradi tega se program, ki uporablja PVM, lahko začne hitreje izvajati. Če je potrebno zagnati veliko kratkih programov, ima PVM prednost, saj je eno samo vzpostavljanje okolja porazdeljeno (amortizirano) čez vse zagone.

Če primerjamo operacije, ki jih podpirata OpenMPI in PVM, opazimo, da PVM ne podpira skupinskih operacij, kot so Alltoall, Allreduce in Allgather. Vse tri operacije imajo predpono **All**, kar pomeni, da je po končani operaciji rezultat na voljo na vseh procesih, in ne samo na enem. OpenMPI ima iz tega vidika prednost, saj moramo te operacije pri PVM ročno implementirati z uporabo bolj osnovnih operacij. Na primer operacijo MPI_Allgather lahko na PVM implementiramo ali z zaporedjem operacij pvm_gather in pvm_bcast ali s tem, da vsak proces pošlje podatke vsakemu drugemu procesu, z operacijama pvm_send in pvm_recv. Seveda je potrebno pri sami implementaciji določiti, katera izmed možnosti je boljša.

Poglavje 5

Problem vzporednih metahevrističnih algoritmov

V tem poglavju si bomo pogledali tri vzporedne metahevristične algoritme, pripravljene za reševanje problema največje neodvisne podmnožice na grafu.

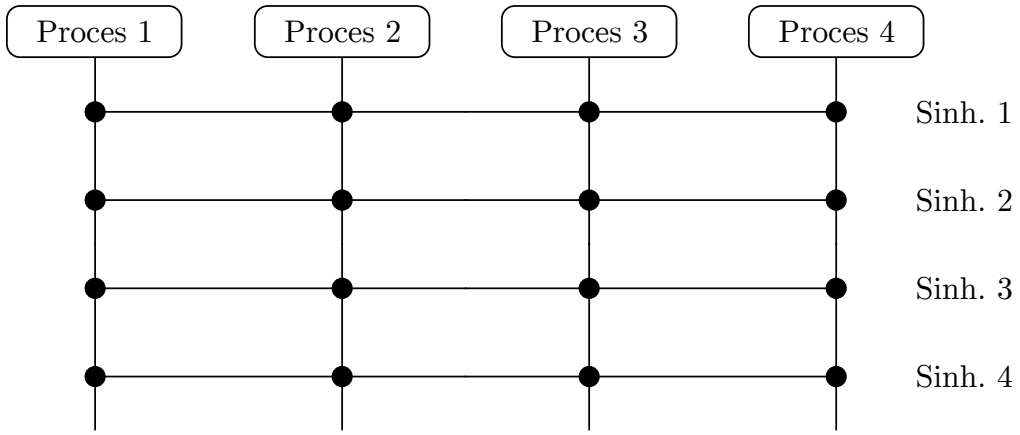
Pri povzporejanju bomo uporabili centraliziran pristop in zato razdelili procese na vodjo in delavce. Eden izmed p procesov bo imel vlogo **vodje**, ostalih $p-1$ procesov pa vlogo **delavca**. Poleg tega delo vodje vključuje tudi vse, kar počnejo delavci. Z drugimi besedami, imamo p procesov, ki opravljajo vlogo delavca, pri čemer eden izmed njih opravlja tudi delo vodje.

Pri vseh vzporednih algoritmih bomo uporabili isti princip za razpošiljanje semen naključnega generatorja (izvede se pred začetkom izvajanja dejanskega algoritma). Vodja preko parametra dobi naključno seme in inicializira svoj naključni generator. Nato generira $p-1$ števil, ki predstavljajo semena za vse ostale procese, delavce. Vsak izmed delavcev dobi od vodje eno seme, s katerim inicializira svoj naključni generator. S tem zagotovimo, da različni delavci različno preiskujejo prostor rešitev.

Pred samim opisom vzporednih algoritmov si oglejmo princip, ki je splošen vsem trem, tj. **sinhronizacija rešitev čez vse procese**. Princip je ponazorjen na sliki 5.1. Sinhronizacijo smo ponazorili z vodoravno črto s polnimi krogci, podobno kot razpošiljanje v [35]. Sinhronizacija se lahko zgodi ob vsaki iteraciji (kot bomo videli pri vzporednih algoritmih razpršeno iskanje in navzkrižna entropija) ali pa na vsake k iteracij (kot pri vzporednem algoritmu simuliranega ohlajanja). Podrobnosti glede izvedbe sinhronizacije si bomo v nadaljevanju pogledali pri opisu vsakega izmed algoritmov.

Oglejmo si še skupinske operacije, ki so skupne vsem trem algoritmom. Zapisali bomo oznake operacij v psevdokodi in njihovo funkcionalnost ter ustrezne

Slika 5.1: Princip sinhronizacije rešitev čez vse procese



OpenMPI in PVM operacije, ki se uporabljajo pri implementaciji.

Zbiranje: pri OpenMPI se uporabi operacija MPI_Gather() in pri PVM operacija pvm_gather().

- Pošlji(ID klica, naslovnik, {podatki}): vsak proces z uporabo zbiranja pošlje svoje podatke naslovniku.
- Zbiranje(ID klica): na naslovniku vrne zbrane podatke, ki so bili poslani od vseh procesov, pri čemer je bil podan enak ID klica.

Razpošiljanje: pri OpenMPI se uporabi operacija MPI_Bcast() in pri PVM operacija pvm_bcast().

- Razpošiljanje(ID klica, {podatki}): natanko en proces pošlje podatke vsem ostalim procesom z uporabo razpošiljanja.
- Sprejmi(ID klica): vrne podatke, ki so bili razposlani s strani določenega drugega procesa, pri čemer je bil podan enak ID klica.

Redukcija: pri OpenMPI se uporabljata operaciji MPI_Reduce() ter MPI_Allreduce() in pri PVM operaciji pvm_reduce() ter nadomestilo za Allreduce (ker PVM nima operacije Allreduce, jo je potrebno nadomestiti z zaporedjem operacij pvm_reduce() in pvm_bcast()).

- Redukcija(operacija, naslovnik, {podatki}): izvede se določena operacija redukcije čez podatke vseh procesov. Opišimo dve operaciji redukcije, ki ju bomo uporabili. Pri operaciji „vsota“ vsak proces prispeva

svojo vrednost k_i za izračun vsote Σk_i, $1 \leq i \leq p$. Pri operaciji „procesMaksimum“ vsak proces prispeva svojo vrednost k_i in nato se določi, kateri proces ima največjo vrednost, $\arg\max_i k_i$, $1 \leq i \leq p$. Kadar se operacija pokliče nad vektorjem, se izvede redukcija komponento po komponento. Rezultat operacije redukcije je dostopen na naslovniku.

- REDUKCIJANAVSE(operacija, {podatki}): deluje enako kot operacija REDUKCIJA, vendar je rezultat dostopen na vseh procesih, in ne samo na naslovniku.

Začeli bomo z opisom vzporedne različice algoritma simulirano ohlajanje in nadaljevali z vzporednima algoritmoma razpršenega iskanja in navzkrižne entropije.

5.1 Vzporedno simulirano ohlajanje

Eden izmed boljših virov s sistematičnim opisom različnih načinov povzporejanja simuliranega ohlajanja je [36]. Opisali so 5 različnih načinov povzporejanja:

1. **Asinhroni pristop**, angl. *asynchronous approach*, pri katerem vsi procesi tečejo neodvisno eden od drugega. Na koncu izvajanja se kot rezultat vrne najboljša najdena rešitev med vsemi procesi.

2. **Sinhroni pristop z občasno izmenjavo rešitev**, angl. *synchronous approach with occasional solution exchanges*, ki je kombinacija simuliranega ohlajanja in genetskega algoritma, pri katerem si delavci posredno, preko vodje, izmenjajo rešitve. Pri posredovanju vodja rešitve med seboj premeša z uporabo genetskega operatorja mutacija.

3. **Sinhroni pristop z občasnim vsiljenjem najboljše rešitve ob fiksnih intervalih**, angl. *synchronous approach with occasional enforcement of best solution – fixed intervals*, pri katerem vodja ob določenih fiksnih intervalih vsem delavcem pošlje najboljšo rešitev, ki je bila najdena do tistega trenutka.

4. **Sinhroni pristop z občasnim vsiljenjem najboljše rešitve ob spremenljivih intervalih**, angl. *synchronous approach with occasional enforcement of best solution – varying intervals*, ki je podoben pristopu iz prejšnje točke, le da je dolžina intervala spremenljiva.

5. **Tesno povezan sinhroni pristop**, angl. *highly coupled synchronous approach*, pri katerem vsak izmed delavcev izračuna po en korak hkrati

in o tem obvesti vodjo. Vodja nato izmed vseh možnih korakov izbere najboljši korak, razpošlje novo rešitev in postopek ponovi.

Poleg petih omenjenih načinov so v [36] razvili tudi hibridni način, ki kombinira točki 3 in 5 ter je bolj učinkovit. Vse načine so preizkusili na veliki množici raznolikih večmodalnih funkcij in pri tem ugotovili, da se je glede kakovosti rešitve najbolje izkazal pristop iz točke 5.

Eno izmed možnih delitev vzporednega simuliranega ohlajanja najdemo v [37], kjer so obravnavali problem dodelitve letal iz flote, angl. *fleet assignment problem*. Osnovna delitev je na en sprehod, angl. *single-walk*, in več sprehodov, angl. *multiple-walks*. Pri tem en sprehod pomeni, da se sledi le eni sami poti, več sprehodov pa pomeni, da se hkrati sledi več potem. Dalje se en sprehod deli na en korak, angl. *single-step*, in več korakov, angl. *multiple-step*. Več sprehodov se deli na interakcijske sprehode, angl. *interacting walks*, in neodvisne sprehode, angl. *independent walks*.

V nadaljevanju si bomo ogledali vzporedni algoritem, ki temelji na povzporedenju algoritma 3.2 s pristopom iz točke 3 vira [36], tj. sinhroni pristop z občasnim vsiljenjem najboljše rešitve ob fiksnih intervalih. Glede na delitev iz vira [37] omenjeni algoritem 3.2 umestimo v kategorijo več korakov, podkategorija interakcijski sprehodi.

Pri vzporednem algoritmu simuliranega ohlajanja (glej algoritem 5.1) delavci na vsakih N_{sinh} iteracij sinhronizirajo najboljšo najdeno rešitev preko vodje. Sinhronizacija poteka tako, da vsak delavec vodji sporoči kakovost rešitve. Naloga vodje je, da izbere delavca, ki ima najboljšo rešitev, in to sporoči vsem delavcem. Izbrani delavec nato razpošlje najboljšo rešitev vsem ostalim in postopek se ponovi. Izvajanje se zaključi, ko je dosežena končna temperatura, podana s parametrom.

Parametri

Parametri α, $c_{zač}$, c_{kon} in L vzporednega algoritma simulirano ohlajanje so definirani enako kot pri zaporednem algoritmu 3.2. Naslednji parameter se uporablja samo pri vzporednem algoritmu.

- N_{sinh}: določa, na vsake koliko število znižanj temperature se naredi sinhronizacija najboljše rešitve preko vodje.

Funkcije

- GenerirajZačetnoRešitev(): generira začetno rešitev, tako da začne s prazno rešitvijo in vključuje eno po eno prosto vozlišče, dokler je to

Algoritem 5.1 Vzporedno simulirano ohlajanje

```
1: function VzporednoSimuliranoOhlajanje(α, c_zač, c_kon, L, N_sinh)
2:     if vodja = taProces then
3:         rešitev ← GenerirajZačetnoRešitev()
4:         Razpošiljanje(1, rešitev)
5:     else
6:         rešitev ← Sprejmi(1)                    // Začetna rešitev enaka vsem
7:     najboljšaRešitev ← rešitev
8:     c ← c_zač
9:     while c > c_kon do
10:        c'_kon ← c × α^N_sinh                   // Končna temp. te sinhronizacije
11:        while c ≥ c'_kon do
12:            for i = 1 to L do
13:                soseda ← Soseda(rešitev)        // Pridobi naključno sosedo
14:                if f(soseda) ≥ f(rešitev) then
15:                    rešitev ← soseda  // Vedno nadomesti z boljšo ali enako
16:                else if exp((f(soseda)−f(rešitev))/c) ≤ rand() then
17:                    rešitev ← soseda
18:                if f(rešitev) ≥ f(najboljšaRešitev) then
19:                    najboljšaRešitev ← rešitev
20:            c ← c · α                           // Zmanjšaj temperaturo
21:        Pošlji(2, vodja, f(najboljšaRešitev))
22:        if vodja = taProces then
23:            kakovosti ← Zbiranje(2)             // Zberi vse kakovosti
24:            izbraniProces ← IzberiNajboljšo(kakovosti)
25:            Razpošiljanje(3, izbraniProces)
26:        izbraniProces ← Sprejmi(3)
27:        if izbraniProces = taProces then
28:            Razpošiljanje(4, rešitev)           // Izbran za razpošiljanje
29:        else
30:            rešitev ← Sprejmi(4)
31:            najboljšaRešitev ← rešitev
32:    if vodja = taProces then
33:        return rešitev
```

mogoče.

- IzberiNajboljšo(kakovosti): vrne identifikacijsko oznako procesa, ki ima najboljšo rešitev. Če jih je več, je eden izmed najboljših naključno izbran.
- Rand(): vrne naključno vrednost iz intervala $[0, 1)$.
- Soseda(rešitev): vrne eno izmed naključno izbranih sosednjih rešitev.

5.2 Vzporedno razpršeno iskanje

Zaporedni algoritem razpršenega iskanja smo že opisali v algoritmu 3.3. Različne načine povzporejanja razpršenega iskanja najdemo v [38], kjer razlikujejo med naslednjimi strategijami posodabljanja referenčne množice (refMnožica): en korak, angl. *single-walk*, ali več korakov, angl. *multiple-walk*. Pri prvi strategiji obstaja le ena sama referenčna množica, ki se razdeli med več procesov zaradi hitrejše obdelave. Druga strategija se deli na dve varianti, neodvisno angl. *independent-threads*, in interaktivno angl. *cooperative-threads*. Pri prvi vsak izmed procesov deluje nad svojo referenčno množico in ne sodeluje z drugimi procesi. Pri drugi varianti si procesi med seboj na različne načine izmenjujejo rešitve.

V nadaljevanju si bomo v algoritmu 5.2 pogledali eno izmed možnih različic povzporedenja, v kateri vsi procesi izvajajo razpršeno iskanje, s tem da sta zbiranje in razpošiljanje referenčne množice prepuščena vodji. Na začetku izvajanja vsak proces generira svojo zalogo naključnih rešitev in pripravi referenčno množico. Nato izvede križanje in mutacijo rešitev referenčne množice ter zbere najboljše in najbolj raznolike rešitve, ki tvorijo novo referenčno množico. Nato vodja zbere referenčne množice vseh procesov in poišče najbolj kakovostne ter najbolj raznolike rešitve. Te shrani v novo referenčno množico, ki jo nato razpošlje vsem procesom. Postopek se ponovi, začenši s korakom, v katerem vsak izmed procesov izvede križanje in mutacijo rešitev referenčne množice.

Parametri

Vsi štirje parametri vzporednega algoritma razpršeno iskanje (p, b_1, b_2 in maksIterBrezIzboljšave) so definirani enako kot pri zaporednem algoritmu 3.3.

Algoritem 5.2 Vzporedno razpršeno iskanje

```
1: function RazpršenoIskanje(p, b1, b2, maksIterBrezIzboljšave)
2:     zaloga ← GenerirajNaključneRešitve(p)
3:     refMnožica ← NajboljšaKakovost(zaloga, b1)
4:     refMnožica ← refMnožica ∪ NajboljšaRazličnost(zaloga,
           refMnožica, b2)
5:     iterBrezIzboljšave ← 0
6:     najboljšaRešitev ← Najboljša(refMnožica)
7:     končaj ← False
8:     while ¬ končaj do
9:         zaloga ← {}
10:        pari ← GenerirajPare(refMnožica)   // Nad svojo ref. množico
11:        for all par ∈ pari do
12:            NarediKombinacijo(par)
13:            zaloga ← zaloga ∪ par
14:        PopraviRešitve(zaloga)
15:        IzboljšajRešitve(zaloga)
16:        IzločiPodvojene(zaloga, refMnožica)
17:        zaloga ← zaloga ∪ refMnožica
18:        refMnožica ← NajboljšaKakovost(zaloga, b1) ∪
               NajboljšaRazličnost(zaloga, refMnožica, b2)
19:        Pošlji(1, vodja, refMnožica)      // Vsak pošlje svojo ref. množico
20:        if vodja = taProces then
21:            zaloga ← Zbiranje(1)          // Vodja zbere vse ref. množice
22:            refMnožica ← NajboljšaKakovost(zaloga, b1)
23:            refMnožica ← refMnožica ∪ NajboljšaRazličnost(zaloga,
                   refMnožica, b2)
24:            if f(Najboljša(refMnožica)) ≥ f(najboljšaRešitev) then
25:                najboljšaRešitev ← Najboljša(refMnožica)
26:                iterBrezIzboljšave ← 0
27:            else
28:                iterBrezIzboljšave++
29:                if iterBrezIzboljšave = maksIterBrezIzboljšave then
30:                    končaj ← True
31:            Razpošiljanje(2, {refMnožica, končaj})
32:        {refMnožica, končaj} ← Sprejmi(2) // Sprejmi novo ref. množico
33:    if vodja = taProces then
34:        return najboljšaRešitev
```

Funkcije

- GenerirajPare(refMnožica): vrne vse možne pare kombinacij rešitev referenčne množice.
- GenerirajRešitve(p): vrne množico p naključno generiranih rešitev. Rešitve so generirane naključno, začenši s prazno rešitvijo in vstavljanjem enega po enega prostega vozlišča, dokler je to mogoče.
- IzboljšajRešitve(zaloga): izvede izboljšavo rešitve. Pri tem se eno po eno vključujejo prosta vozlišča, dokler je to mogoče.
- IzločiPodvojene(zaloga, refMnožica): iz zaloge izloči vse takšne rešitve, ki so že prisotne v referenčni množici.
- Najboljša(refMnožica): vrne najboljšo rešitev glede na kakovost.
- NajboljšaKakovost(zaloga, k): vrne k najboljših rešitev iz zaloge in jih pri tem izloči iz zaloge.
- NajboljšaRazličnost(zaloga, refMnožica, k): vrne k najbolj različnih rešitev iz zaloge v primerjavi z rešitvami v refMnožica in jih pri tem izloči iz zaloge. Različnost se meri kot evklidska razdalja med rešitvami čez vse kombinacije rešitev v zalogi in referenčni množici. Najbolj različne se poiščejo s pomočjo max-min kriterija (razdelek 2 v [16]).
- NarediKombinacijo(par): izvede kombinacijo med dvema rešitvama z enakomernim naključnim križanjem, angl. *uniform random crossover*, vozlišče po vozlišče.
- PopraviRešitve(zaloga): za vsako rešitev v zalogi izvede popravilo vseh nesprejemljivih rešitev. Pri tem eno po eno izloča nesprejemljiva vozlišča v naključnem vrstnem redu, dokler rešitev ne postane sprejemljiva.

5.3 Vzporedna navzkrižna entropija

Uporaba vzporednega algoritma za navzkrižno entropijo (glej algoritem 5.3) je opisana v [39]. Razpoznali so dve nalogi, ki ju lahko povzporedimo. Prva naloga je generiranje vzorcev na podlagi verjetnosti. Njihova ideja je bila, da vsak izmed p procesov namesto N vzorcev generira $\frac{N}{p}$ vzorcev. Druga naloga je posodabljanje verjetnosti na podlagi generiranih vzorcev, ki jo je mogoče dalje deliti na ocenjevanje vzorcev in posodabljanje verjetnosti. Pri njihovi implementaciji posodabljanje verjetnosti ni bilo vzporedno, ampak posredno (centralizirano) preko vodje. Glede ocenjevanja rešitev je bila njihova ideja ta, da vsak izmed procesov oceni tiste rešitve (oz. del vzorca), ki jih je sam

generiral.

Uporaba vzporednega algoritma za navzkrižno entropijo za reševanje problema največjega polnega podgrafa je opisana v [21]. Njihov algoritem uporablja MPI in je podoben tistemu v [39], vendar vodje niso uporabili za posodabljanje verjetnosti, ampak za računanje vrednosti γ, ki določa prag minimalne kakovosti rešitve, da se določen vzorec uporabi za posodobitev verjetnosti. Med drugim so za zbiranje najboljših rešitev uporabili skupinsko operacijo MPI_Allreduce.

V nadaljevanju si bomo ogledali povzporedenje, ki uporablja podobne principe kot [21, 39]. Razlika je v tem, da ne bomo uporabljali izmenjevanja rešitev, ampak samo izmenjevanje podatkov o verjetnostni matriki. Vsak proces izračuna svoj vzorec in na njegovi podlagi pripravi matriko verjetnosti. Nato se naredi skupinska operacija Allreduce z uporabo vsote čez vse procesorje. Tako ima vsak izmed procesorjev enako matriko verjetnosti, ki je rezultat vseh procesorjev. Pri OpenMPI v ta namen uporabimo operacijo MPI_Allreduce. Ker PVM ne podpira operacije redukcije, ki bi rezultat pustila na vseh procesih, uporabimo centraliziran pristop (najprej redukcija na enega izmed procesov, ki ji sledi razpošiljanje).

Parametri

Vsi štirje parametri (ρ, c, α in maksIterBrezIzboljšave) vzporednega algoritma metode navzkrižne entropije so definirani enako kot pri zaporednem algoritmu 3.4.

Funkcije

- Dodaj(urejeniSeznam, rešitev): doda rešitev v urejeni seznam.
- GenerirajRešitev(V): vrne novo rešitev, ki je generirana s pomočjo Bernoullijeve porazdelitve z uporabo verjetnosti, ki so podane v matriki V.
- PosodobiVerjetnosti(V, α, urejeniSeznam): posodobi matriko verjetnosti V glede na rešiteve v urejenem seznamu. Parameter α določa glajenje, tako da je α-delež nove verjetnosti združen z $(1 - \alpha)$-deležem stare verjetnosti za določeno vozlišče.
- Prva(urejeniSeznam): vrne najbolj kakovostno rešitev na seznamu.
- Skrajšaj(urejeniSeznam, N_{najb}): v urejenem seznamu zadrži samo N_{najb} najboljših rešitev, ostale zanemari.

Algoritem 5.3 Vzporedna navzkrižna entropija

```
1: function VzporednaNavzkrižnaEntropija(ρ, c, α)
2:     V ← {{0.5, ..., 0.5}, ..., {0.5, ..., 0.5}}          // Matrika verjetnosti
3:     N ← n × c                                              // Velikost vzorca
4:     N_najb ← N × ρ                  // Število rešitev vzorca za posodobitev
5:     urejeniSeznam ← {}                  // Rešitve razvrščene po kakovosti
6:     iterBrezIzboljšave ← 0
7:     najboljšaRešitev ← Najboljša(refMnožica)
8:     končaj ← 0
9:     while končaj < p do                  // Dokler vsi procesi ne končajo
10:        for i = 1 to N do                    // Generiraj vzorčne rešitve
11:            rešitev ← GenerirajRešitev(V)
12:            Dodaj(urejeniSeznam, rešitev)      // Dodaj rešitev v seznam
13:        Skrajšaj(urejeniSeznam, N_najb)   // Uporabi samo N_najb rešitev
14:        končaj ← 0
15:        if f(Prva(refMnožica)) ≥ f(najboljšaRešitev) then
16:            najboljšaRešitev ← Najboljša(refMnožica)
17:            iterBrezIzboljšave ← 0
18:        else
19:            iterBrezIzboljšave++
20:            if iterBrezIzboljšave ≥ maksIterBrezIzboljšave then
21:                končaj ← 1
22:        V ← PosodobiVerjetnosti(V, α, urejeniSeznam)
23:        {V, končaj} ← RedukcijaNaVse(vsota, {V, končaj})
24:    izbraniProces ← RedukcijaNaVse(procesMaksimum,
           f(Prva(urejeniSeznam)))
25:    if izbraniProces = taProces then
26:        return Prva(urejeniSeznam)      // Najboljša rešitev izbranega
                                               procesa
```

5.4 Smiselnost povzporejanja

Pri povzporejanju metahevrističnega algoritma se lahko srečamo s problemom, da je rezultat vzporedne različice algoritma relativno slab. Razlogov za to je več, eden izmed pomembnejših je, da smo s povzporedenjem povečali izostritev, pri čemer je razpršitev ostala nespremenjena. To ima lahko za posledico, da algoritem hitreje obtiči v lokalnem optimumu, iz katerega več ne more pobegniti.

Če želimo ugotoviti, kdaj je povzporedenje določenega algoritma smiselno, si zamislimo sledeč kriterij. Rekli bomo, da je povzporedenje smiselno takrat, kadar je vzporedni metahevristični algoritem boljši od naivnega vzporednega algoritma. S pojmom naivni vzporedni algoritem označimo hkratni zagon ekvivalentnega zaporednega algoritma na več procesih. Če torej enkrat zaženemo vzporedni algoritem na p procesih in dobimo rezultat f_{vzp} ter p-krat zaženemo zaporedni algoritem na enakem računalniku in označimo najboljšega izmed p rezultatov z f_{naiv}, mora biti f_{vzp} v povprečju, čez raznolike primerke problema, vsaj tako dober kot f_{naiv}. V nasprotnem primeru težko upravičimo povzporejanje.

Poglavje 6

Preizkusi

6.1 Preizkusna okolja

V svojih preizkusih smo uporabili štiri različne sisteme: dva različna Dellova strežnika z navideznima strojema VMware, heterogeni sistem Linux in FreeBSD strežnikov ter superračunalnik LSC Adria.

6.1.1 Strežnik Dell 1950

Na strežniku Dell PowerEdge 1950, na kratko Dell 1950, je nameščen navidezni stroj VMware ESX 3.5.0. Strežnik ima en dvojedrni procesor Intel Xeon 5130 (frekvenca 2.00 GHz, predpomnilnik L2 2x2 MB), pomnilnik 4 GB na 1333MHz vodilu. Ker sta na razpolago dve jedri, lahko na tem strežniku učinkovito izvajamo do največ dva procesa hkrati.

V navideznem stroju je bil postavljen navidezni računalnik z operacijskim sistemom Linux Ubuntu 9.04, arhitektura x86, strežniška različica s podporo za SMP, jedro 2.6.28-15-server. Pri prevajanju algoritmov je bil uporabljen GCC 4.3.3.

6.1.2 Strežnik Dell R200

Na strežniku Dell PowerEdge R200, na kratko Dell R200, je nameščen navidezni stroj VMware ESXi 4.0.0. Strežnik ima en štirijedrni procesor Intel Xeon X3220 (frekvenca 2.40 GHz, predpomnilnik L2 2x4 MB), pomnilnik 8 GB na 1066MHz vodilu. Ker so na razpolago štiri jedra, lahko na tem strežniku učinkovito izvajamo do največ štiri procese hkrati.

Tabela 6.1: Strežniki heterogenega sistema

Strežnik	Operac. sistem	Procesor	Pomnilnik
1	Ubuntu 8.10	Intel Pentium 4, 2.80 GHz	512 MB
2	Ubuntu 8.10	Intel Pentium 4, 2.40 GHz	512 MB
3	FreeBSD 6.2	Intel Pentium 4, 2.40 GHz	512 MB
4	FreeBSD 6.2	Intel Celeron, 2.80 GHz	1024 MB

V navideznem stroju VMware so bili postavljeni štirje identični navidezni računalniki, ki so tvorili vzporedni stroj. Opis vsakega izmed njih je enak tistemu, ki je opisan pri strežniku Dell 1950. Pri izvedbi preizkusov vzporednih algoritmov sta bila uporabljena OpenMPI 1.3 in PVM 3.4.5 (navodila za namestitev najdemo v dodatku A).

6.1.3 Heterogeni sistem strežnikov

Sestava gruče heterogenih računalnikov je razvidna iz tabele 6.1. V uporabi sta operacijska sistema Linux Ubuntu 8.10 in FreeBSD 6.2 na procesorjih Intel Pentium 4 in Intel Celeron. Vidimo, da je takšna gruča strežnikov heterogena v smislu operacijskih sistemov in procesorjev, ne pa tudi arhitektur, saj imata oba procesorja x86 arhitekturo. Strežnika, na katerih je nameščen Linux Ubuntu, sta uporabljala jedro 2.6.27-9-generic in strežnik z operacijskim sistemom FreeBSD jedro 6.2-RELEASE. Pri prevajanju algoritmov na FreeBSD strežnikih je bil uporabljen GCC 3.4.6 in na Linux Ubuntu strežnikih GCC 4.3.2. Pri izvedbi preizkusov vzporednih algoritmov sta bila na obeh operacijskih sistemih uporabljena OpenMPI 1.3.3 in PVM 3.4.5 (navodila za namestitev najdemo v dodatku A).

Kljub heterogenosti gruče sta tako OpenMPI kot PVM delovala brez težav (s tem da moramo pri OpenMPI izrecno vključiti podporo za heterogene sisteme). Rezultatov preizkusov na tem sistemu ne bomo prikazali, saj je bil namen preizkusov le potrditi, da je mogoče OpenMPI in PVM uporabiti tudi na heterogenih okoljih. Kljub temu da je v splošnem izvajanje na heterogenih okoljih nekoliko počasnejše (zaradi potrebnih pretvorb iz zapisa debelega konca v zapis tankega konca ipd.), je običajno še vedno bolje bolje uporabiti vzporedni algoritem, četudi na heterogenem okolju, kot pa sploh ne.

6.1.4 Superračunalnik LSC Adria

Superračunalnik Ljubljana Supercomputing Center – LSC Adria je v lasti podjetja Turboinštitut, d.d., Ljubljana, in se uporablja predvsem za računanje dinamike tekočin, angl. *computational fluid dynamics*. Sestavljen je iz 256 strežnikov (nameščenih v enote IBM BladeCenter H), vsak izmed strežnikov vsebuje dva štirijedrna procesorja Intel Xeon E5530 (frekvenca 2.40 GHz, predpomnilnik L2 4x256 KB in L3 8 MB), pomnilnik 16 GB na 1066MHz vodilu. Za povezavo med strežniki skrbita dve gigabitni LAN omrežji in eno InfiniBand omrežje. Na vsakemu od strežnikov je bil nameščen operacijski sistem Linux CentOS 4.6, arhitektura x86-64, strežniška različica s podporo za SMP, jedro 2.6.9-67.ELlargesmp. Pri prevajanju algoritmov je bil uporabljen GCC 3.4.6. Pri izvedbi preizkusov vzporednih algoritmov sta bila uporabljena OpenMPI 1.2.6 in PVM 3.4.5 (oba sta bila že predhodno nameščena na superračunalniku, navodila za namestitev najdemo v dodatku A).

Superračunalnik je z LINPACK primerjalnim preizkusom dosegel Rmax = 35.08 TFlops in Rpeak = 49.15 TFlops. Na tej podlagi je bil junija leta 2008 uvrščen na 53. mesto lestvice svetovnih superračunalnikov TOP500 [40].

6.2 DIMACS primerki

DIMACS primerki so bili leta 1993 pripravljeni za tretji DIMACS izziv z naslovom NP-težki problemi: največji poln podgraf, barvanje grafa in izpolnjivost, angl. *NP Hard Problems: Maximum Clique, Graph Coloring, and Satisfiability* [41], in se še danes pogosto uporabljajo za preizkus metahevrističnih algoritmov. Primerki so prosto dostopni na spletu [42]. DIMACS primerki so sicer v prvi vrsti namenjeni reševanju problema največjega polnega podgrafa. Ker je reševanje tega problema ekvivalentno reševanju problema največje neodvisne množice (kot opisano v razdelku 2.6.4), lahko vse grafe primerkov pretvorimo na komplementarne grafe in na njih rešujemo problem največje neodvisne množice.

Na voljo je 80 različnih primerkov. Zaporedne algoritme bomo preizkusili na vseh 80 primerkih. Ker je bil čas izvajanja na superračunalniku omejen, smo preizkuse vzporednih algoritmov izvedli na sedmih primerkih, izbranih glede na velikost (tj. število vozlišč grafa določenega primerka). Izbrali smo primerke brock200_1, C1000.9, C4000.5, keller6, MANN_a9, MANN_a45 in MANN_a81. S temi sedmimi primerki je precej dobro pokrita množica vseh 80 DIMACS primerkov, saj z njimi pokrijemo celoten razpon števila vozlišč, od 45 vozlišč pri primerku MANN_a9 do 4000 vozlišč pri primerku C4000.5. Ostalih pet

Tabela 6.2: DIMACS primerjalni preizkus

Primerek	**Čas (sekunde)**		
	Dell 1950	**Dell R200**	**LSC Adria**
r100.5	$< \epsilon$	$< \epsilon$	$< \epsilon$
r200.5	$0,034$	$0,03$	$0,03$
r300.5	$0,372$	$0,235$	$0,29$
r400.5	$2,456$	$1,406$	$1,829$
r500.5	$10,31$	$5,332$	$6,959$

primerkov je razvrščenih znotraj teh skrajnosti.

6.2.1 DIMACS primerjalni preizkus

Na obeh strežnikih Dell in na LSC Adria je bil pognan DIMACS primerjalni preizkus, angl. *benchmark*, ki se uporablja za določitev zmogljivosti računalnikov, z namenom primerjave rezultatov med različnimi avtorji, ki uporabljajo različne računalnike. V tabeli 6.2 so prikazani časi na petih primerkih, ki se običajno uporabljajo pri DIMACS primerjalnemu preizkusu [41]. Vse meritve so bile opravljene 10-krat in povprečene, za boljšo natančnost in zanesljivost.

6.3 Preizkusni primeri

Zaporedne in vzporedne različice metahevrističnih algoritmov simulirano ohlajanje, razpršeno iskanje in navzkrižna entropija smo preizkusili na problemu največje neodvisne množice. Vseh šest algoritmov (s številkami 3.2, 3.3, 3.4, 5.1, 5.2 in 5.3) je bilo implementiranih v programskem jeziku C++. Pri preizkusih smo uporabili DIMACS primerke.

Vse meritve so bile opravljene z uporabo 10 različnih semen za naključni generator (razen kjer bomo izrecno navedli, da je bilo uporabljeno različno število semen). Izpostaviti moramo, da je imel OpenMPI pri preizkusih na superračunalniku LSC Adria na razpolago omrežje InfiniBand, PVM pa ne, saj ga ne podpira. Pri preizkusih tako zaporednih kot vzporednih algoritmov so bili uporabljeni parametri, navedeni v tabeli 6.3. Ti parametri so bili določeni eksperimentalno z namenom zagotavljanja relativno dobrih rešitev pri sprejemljivem času izvajanja.

Tabela 6.3: Parametri pri preizkusih

Algoritem	**Parameter**	**Vrednost**
Simulirano ohlajanje	α	0.9995
	$c_{zač}$	100
	c_{kon}	1
	L	100
	N_{sinh} *(vzporedni alg.)*	500
Razpršeno iskanje	p	400
	b_1	20
	b_2	20
	maksIterBrezIzboljšave	20
Navzkrižna entropija	ρ	0.1
	c	0.1
	α	0.5
	maksIterBrezIzboljšave	10

Na podlagi rezultatov preizkusov si bomo v naslednjem poglavju pogledali sledeče primerjave:

- Uporaba vseh 80 DIMACS primerkov:
  - Primerjava treh zaporednih algoritmov (Dell 1950).
  - Kakovost algoritma simulirano ohlajanje (Dell 1950).
- Uporaba izbranih sedmih DIMACS primerkov:
  - Cena komunikacije (LSC Adria in Dell R200).
  - Čas izvajanja in število iteracij (LSC Adria in Dell R200).
  - Kakovost rešitev in število iteracij (neodvisno od okolja).
  - Profili kakovosti rešitev skozi iteracije (neodvisno od okolja).

Poglavje 7

Rezultati

V tem poglavju si bomo ogledali rezultate preizkusov in primerjav. Za prikaz rezultatov bomo v veliki meri uporabljali tabele, zato se moramo najprej seznaniti z notacijo. Kadar imamo več zagonov z različnimi semeni (v našem primeru običajno 10), dobimo toliko različnih meritev. Za ponazoritev takšnega rezultata v tabelah bomo uporabili sledečo notacijo: „povprečje (minimum, maksimum)“, kjer povprečje predstavlja navadno povprečje rezultatov, minimum predstavlja najmanjšega izmed rezultatov in maksimum največjega. Poleg tega moramo pojasniti še, na kakšen način so bili pridobljeni rezultati. Če ni drugače zapisano, rezultati odražajo kakovost najboljše dosežene rešitve in čas (v sekundah), v katerem je bila ta najboljša rešitev prvič dosežena. Tudi če algoritem naleti na rešitev enake kakovosti ponovno kasneje, nas ta čas ne zanima.

Omeniti velja oznake v primerih, ko rezultatov ni bilo mogoče pridobiti ali izračunati. V primeru, da določena meritev ni bila opravljena (npr. zaradi pomanjkanja časa), bomo to označili z *N.P.*, kar pomeni „ni podatka“. Pri določenih meritvah časa se lahko zgodi, da je čas manjši od natančnosti merjenja in zato ni mogoče zapisati točne vrednosti. V takih primerih bomo to označili z $< \epsilon$. Kadar določenega izraza (npr. ulomka) ni mogoče ovrednotiti, ker je ena izmed vrednosti enaka nič ali manjša od ϵ, bomo to označili z *N.M.I.*, kar pomeni „ni mogoče izračunati“.

7.1 Primerjava treh zaporednih algoritmov

V tabeli 7.2 vidimo primerjavo doseženih rešitev in časov za tri zaporedne metahevristične algoritme: simulirano ohlajanje, razpršeno iskanje in navzkrižna entropija. Preizkus je bil izveden na vseh 80 DIMACS primerkih za problem

Tabela 7.1: Uvrstitev treh zaporednih algoritmov za problem največje neodvisne množice (Dell 1950)

(a) Glede na kakovost rešitve

Algoritem	1. mesto	2. mesto	3. mesto
Simulirano ohlajanje	66	4	10
Razpršeno iskanje	37	43	0
Navzkrižna entropija	17	1	62

(b) Glede na čas izvajanja

Algoritem	1. mesto	2. mesto	3. mesto
Simulirano ohlajanje	43	24	13
Razpršeno iskanje	9	32	39
Navzkrižna entropija	29	26	25

največje neodvisne množice (kot je opisan v razdelku 2.6.1). Ugotovimo lahko, da algoritem simulirano ohlajanje doseže enako dober ali boljši rezultat kot ostala dva algoritma pri vseh primerkih, razen pri hamming8-2, hamming10-2 in MANN_a81.

Rezultate dosežene povprečne kakovosti in časov izvajanja lahko analiziramo s pomočjo uvrščanja algoritmov, angl. *ranking*. Za vsakega od 80 primerkov razdelimo prvo, drugo in tretje mesto glede na doseženo kakovost rešitve in glede na čas izvajanja posameznega algoritma. V tabeli 7.1 vidimo, kolikokrat je določen algoritem dosegel prvo, drugo ali tretje mesto, sešteto čez vse primerke. Seštevek po stolpcih ni nujno enak 80, ker imata lahko dva ali celo vsi trije algoritmi pri določenem primerku enak rezultat in jim je zato potrebno dodeliti isto mesto.

Za algoritem simulirano ohlajanje smo že ugotovili, da je najboljši, in tabela 7.1a to potrjuje. Ugotovimo lahko tudi, da je v splošnem razpršeno iskanje boljše od navzkrižne entropije, vendar razlika ni tako velika kot med simuliranim ohlajanjem in razpršenim iskanjem. Če primerjamo še uvrščanje časov, v tabeli 7.1b vidimo, da je simulirano ohlajanje največkrat zasedlo prvo mesto, sledi mu navzkrižna entropija in na koncu razpršeno iskanje.

Tabela 7.2: Primerjava treh zaporednih algoritmov za problem največje neodvisne množice (Dell 1950)

Primerek	Simulirano ohlajanje		Razpršeno iskanje		Navzkrižna entropija	
	Kakovost	Čas (s)	Kakovost	Čas (s)	Kakovost	Čas (s)
brock200_1	21	2,608 (0,659; 3,367)	20,3 (20; 21)	0,339 (0,093; 1,805)	18,3 (18; 20)	0,076 (0,019; 0,19)
brock200_2	12	0,128 (0,032; 0,281)	11,9 (11; 12)	0,439 (0,022; 1,248)	9,8 (9; 10)	0,066 (0,033; 0,122)
brock200_3	15	0,141 (0,029; 0,321)	14,2 (14; 15)	0,202 (0,097; 0,396)	12,4 (12; 13)	0,076 (0,03; 0,131)
brock200_4	17	1,368 (0,448; 2,585)	16,3 (16; 17)	0,371 (0,02; 0,967)	14,2 (13; 15)	0,074 (0,029; 0,122)
brock400_1	25	5,2 (4,044; 5,766)	24,4 (24; 25)	1,506 (0,747; 3,361)	21,3 (20; 22)	1,124 (0,272; 2,081)
brock400_2	25,3 (25; 28)	5,32 (3,886; 6,098)	24,2 (24; 25)	1,382 (0,554; 3,238)	22 (20; 24)	1,57 (0,171; 2,393)
brock400_3	26,3 (25; 30)	4,296 (2,752; 5,465)	24,1 (23; 25)	1,128 (0,2; 1,919)	21,5 (20; 23)	0,936 (0,273; 1,714)
brock400_4	29,1 (25; 33)	5,384 (1,458; 6,407)	24,1 (23; 25)	1,6 (0,716; 3,444)	21,9 (20; 24)	1,128 (0,146; 2,087)
brock800_1	20,4 (20; 21)	11,627 (1,483; 15,63)	19,7 (19; 20)	3,697 (0,879; 8,41)	17,8 (17; 19)	7,893 (1,182; 20,932)
brock800_2	20,7 (20; 21)	13,582 (10,92; 16,951)	19,9 (19; 21)	3,97 (1,589; 9,734)	17,8 (17; 19)	5,195 (2,24; 14,375)
brock800_3	21,2 (20; 22)	13,209 (11,083; 14,475)	20,1 (19; 21)	4,186 (1,609; 7,562)	17,5 (17; 19)	4,747 (1,197; 13,686)
brock800_4	20,6 (20; 21)	13,137 (9,318; 15,23)	19,6 (19; 20)	3,117 (0,927; 8,6)	17,8 (17; 19)	5,089 (1,156; 11,003)
C125.9	34	0,687 (0,332; 0,97)	34	0,102 (0,052; 0,124)	33,4 (33; 34)	0,019 (0,013; 0,028)
C250.9	44	1,879 (1,64; 2,065)	43,7 (43; 44)	0,555 (0,227; 1,467)	42 (41; 43)	0,47 (0,282; 0,758)
C500.9	56,2 (56; 57)	4,093 (3,443; 5,128)	52,8 (51; 54)	1,365 (1,094; 1,578)	52,2 (46; 54)	5,358 (1,01; 7,518)
C1000.9	66,8 (65; 68)	7,676 (6,154; 10,535)	61,7 (58; 64)	5,813 (3,519; 14,8)	59,4 (52; 64)	47,903 (2,019; 82,363)
C2000.5	15,9 (15; 16)	40,907 (3,341; 52,638)	15	8,7 (2,568; 14,68)	14,1 (14; 15)	94,192 (48,143; 152,246)
C2000.9	74,9 (74; 76)	17,221 (14,204; 21,053)	67,2 (64; 69)	13,544 (7,642; 25,604)	59 (58; 60)	101,943 (15,705; 248,642)
C4000.5	16,7 (16; 17)	67,926 (12,73; 111,705)	15,6 (15; 16)	18,609 (6,134; 40,511)	15	443,785 (121,571; 1299,663)
c-fat200-1	12	0,034 (0,024; 0,059)	12	0,046 (0,038; 0,075)	12	0,042 (0,035; 0,054)
c-fat200-2	22,2 (22; 24)	0,031 (0,023; 0,055)	24	0,055 (0,045; 0,096)	24	0,052 (0,033; 0,138)
c-fat200-5	56,8 (56; 58)	0,027 (0,016; 0,042)	58	0,063 (0,059; 0,065)	58	0,034 (0,028; 0,042)
c-fat500-1	14	0,216 (0,165; 0,405)	14	0,216 (0,203; 0,262)	13,8 (13; 14)	0,473 (0,395; 0,71)
c-fat500-2	25,2 (24; 26)	0,205 (0,158; 0,388)	26	0,244 (0,228; 0,299)	26	0,42 (0,4; 0,434)
c-fat500-5	62,4 (62; 64)	0,183 (0,141; 0,349)	64	0,335 (0,287; 0,547)	64	0,434 (0,386; 0,584)
c-fat500-10	125,3 (124; 126)	0,16 (0,112; 0,465)	126	0,353 (0,322; 0,42)	126	0,431 (0,388; 0,539)
DSJC500.5	13	4,121 (0,525; 8,359)	12,8 (12; 13)	1,059 (0,294; 2,211)	11,5 (11; 12)	1,19 (0,332; 2,696)
DSJC1000.5	14,5 (14; 15)	13,638 (1,342; 26,532)	13,9 (13; 14)	2,954 (0,787; 6,654)	12,9 (12; 14)	7,8 (2,325; 23,378)
gen200_p0.9_44	44	1,637 (1,409; 1,999)	43,6 (41; 44)	0,42 (0,281; 1,187)	40,4 (37; 44)	0,219 (0,114; 0,39)
gen200_p0.9_55	55	1,637 (1,209; 1,862)	55	0,181 (0,089; 0,241)	49,6 (40; 55)	0,141 (0,079; 0,237)
gen400_p0.9_55	55	3,304 (2,773; 3,923)	50,7 (50; 52)	1,027 (0,865; 1,215)	49,3 (48; 51)	2,685 (1,957; 3,984)
gen400_p0.9_65	65	3,434 (2,493; 4,513)	64,7 (62; 65)	1,017 (0,718; 1,589)	56,8 (48; 65)	3,015 (2,009; 4,336)
gen400_p0.9_75	75	3,871 (3,689; 4,159)	75	0,633 (0,55; 0,744)	66,4 (51; 75)	2,658 (1,637; 4,433)
hamming6-2	31,2 (31; 32)	0,674 (0,001; 1,207)	32	0,007 (0,004; 0,011)	32	0,006 (0,002; 0,011)
hamming6-4	4	0,007 (0,003; 0,014)	4	0,009 (0,005; 0,012)	4	0,007 (0,003; 0,012)
hamming8-2	127	2,022 (1,991; 2,076)	128	0,115 (0,108; 0,153)	128	0,23 (0,187; 0,265)
hamming8-4	16	0,082 (0,023; 0,178)	16	0,107 (0,027; 0,119)	15,1 (13; 16)	0,164 (0,042; 0,345)
hamming10-2	511	5,641 (5,216; 6,205)	512	1,354 (0,97; 1,813)	512	31,803 (29,484; 33,136)
hamming10-4	40	10,374 (10; 10,864)	36,9 (35; 40)	6,392 (2,963; 14,351)	35 (32; 40)	49,105 (10,839; 82,819)
johnson8-2-4	4	0,007 (0,001; 0,028)	4	0,006 (0,002; 0,01)	4	0,005 (0,001; 0,012)

(Se nadaljuje)

Tabela 7.2: Primerjava treh zaporednih algoritmov za problem največje neodvisne množice (Dell 1950, nadaljevanje)

Primerek	Simulirano ohlajanje		Razpršeno iskanje		Navzkrižna entropija	
	Kakovost	Čas (s)	Kakovost	Čas (s)	Kakovost	Čas (s)
johnson8-4-4	14	0,006 (0,002; 0,01)	14	0,008 (0,004; 0,011)	14	0,007 (0,002; 0,015)
johnson16-2-4	8	0,008 (0,003; 0,017)	8	0,011 (0,007; 0,015)	8	0,009 (0,005; 0,018)
johnson32-2-4	16	0,035 (0,022; 0,077)	16	0,04 (0,036; 0,046)	16	0,278 (0,265; 0,289)
keller4	11	0,028 (0,009; 0,093)	11	0,033 (0,013; 0,175)	11	0,033 (0,015; 0,072)
keller5	27	8,154 (7,313; 8,794)	26,6 (26; 27)	2,126 (0,794; 4,485)	24,2 (23; 26)	9,731 (1,094; 16,921)
keller6	57,4 (55; 59)	33,437 (23,748; 41,559)	51,2 (48; 55)	23,738 (11,991; 39,763)	45,7 (43; 50)	571,177 (71,586; 1846,382)
MANN_a9	16	0,006 (0,001; 0,022)	16	0,006 (0,003; 0,011)	16	0,011 (0,001; 0,033)
MANN_a27	125,8 (125; 126)	4,097 (3,154; 5,332)	125,2 (124; 126)	0,719 (0,014; 1,167)	123,5 (123; 124)	0,46 (0,123; 0,863)
MANN_a45	338,9 (338; 340)	11,159 (10,299; 11,985)	338,1 (337; 339)	2,256 (0,038; 5,177)	336,9 (336; 338)	12,616 (2,405; 29,592)
MANN_a81	1082,8 (1081; 1085)	0,084 (0,055; 0,265)	1087,3 (1087; 1088)	4,986 (0,159; 26,005)	1088,6 (1088; 1090)	378,69 (80,874; 734,291)
p_hat300-1	8	0,146 (0,072; 0,339)	8	0,139 (0,065; 0,286)	7,2 (7; 8)	0,156 (0,089; 0,407)
p_hat300-2	25	1,865 (0,242; 2,898)	25	0,243 (0,165; 0,329)	24,2 (23; 25)	0,423 (0,218; 0,651)
p_hat300-3	36	2,844 (2,345; 3,41)	35,8 (34; 36)	0,487 (0,282; 0,92)	33,9 (33; 36)	0,645 (0,404; 1,213)
p_hat500-1	9	0,271 (0,141; 0,583)	9	0,292 (0,151; 0,332)	8,3 (8; 9)	0,654 (0,354; 1,56)
p_hat500-2	36	4,319 (2,649; 5,059)	36	0,755 (0,573; 0,932)	34,7 (31; 36)	2,27 (1,347; 3,564)
p_hat500-3	50	4,198 (3,757; 4,703)	49,4 (49; 50)	0,831 (0,496; 1,201)	48,8 (47; 50)	3,995 (2,325; 6,549)
p_hat700-1	11	8,425 (1,645; 17,681)	11	1,515 (1,108; 3,777)	8,7 (8; 9)	3,042 (0,978; 6,627)
p_hat700-2	44	6,353 (5,251; 7,027)	44	1,189 (0,897; 1,59)	43,5 (42; 44)	8,59 (6,873; 11,001)
p_hat700-3	62	4,452 (3,932; 5,25)	61,9 (61; 62)	1,571 (1,183; 2,231)	60,9 (60; 62)	11,402 (7,244; 15,085)
p_hat1000-1	10	2,376 (0,832; 5,9)	10	1,19 (0,938; 1,482)	9,2 (9; 10)	9,167 (2,455; 23,585)
p_hat1000-2	46	9,536 (7,977; 11,394)	46	2,645 (1,477; 8,01)	45,6 (45; 46)	31,238 (20,958; 45,509)
p_hat1000-3	68	7,877 (6,135; 8,687)	65,6 (64; 68)	2,595 (1,647; 3,638)	65 (64; 68)	46,363 (24,067; 62,039)
p_hat1500-1	11,8 (11; 12)	36,515 (1,478; 52,699)	11,1 (11; 12)	4,123 (1,847; 7,222)	10,1 (10; 11)	24,125 (7,768; 60,19)
p_hat1500-2	65	13,701 (11,894; 14,951)	64,7 (64; 65)	5,039 (4,154; 6,506)	64,4 (61; 65)	99,521 (66,331; 162,967)
p_hat1500-3	94	10,444 (9,235; 11,855)	93 (92; 94)	4,178 (3,466; 5,63)	92,9 (92; 93)	96,355 (81,321; 111,768)
san200_0.7_1	29 (21; 30)	3,75 (0,085; 5,295)	30	0,082 (0,019; 0,122)	26,4 (16; 30)	0,09 (0,021; 0,14)
san200_0.7_2	16,7 (15; 18)	2,714 (0,026; 4,612)	18	0,524 (0,25; 1,185)	14,8 (14; 17)	0,062 (0,019; 0,151)
san200_0.9_1	69,6 (69; 70)	2,402 (1,96; 2,982)	70	0,141 (0,097; 0,212)	67,3 (51; 70)	0,114 (0,097; 0,152)
san200_0.9_2	60	2,391 (2,235; 2,508)	60	0,153 (0,089; 0,213)	52,5 (40; 60)	0,134 (0,016; 0,187)
san200_0.9_3	44	1,836 (1,634; 1,981)	43,2 (36; 44)	0,339 (0,268; 0,456)	35,3 (35; 36)	0,163 (0,131; 0,229)
san400_0.5_1	13	1,313 (0,166; 3,094)	12,6 (9; 13)	0,44 (0,201; 1,097)	8,4 (8; 9)	0,446 (0,176; 1,54)
san400_0.7_1	23,6 (21; 40)	5,401 (0,034; 9,271)	40	0,477 (0,239; 1,074)	25,6 (22; 40)	0,382 (0,175; 1,128)
san400_0.7_2	20,2 (19; 30)	7,945 (0,531; 10,033)	30	1,413 (0,397; 3,062)	17,9 (17; 19)	0,527 (0,161; 0,983)
san400_0.7_3	18,9 (17; 22)	7,351 (0,339; 8,972)	17,5 (17; 18)	0,837 (0,182; 2,147)	15,8 (15; 16)	0,55 (0,277; 0,894)
san400_0.9_1	100	4,652 (4,005; 5,602)	100	0,419 (0,205; 0,711)	78,9 (54; 100)	1,212 (0,127; 3,449)
san1000	11,3 (10; 15)	3,693 (0,385; 16,656)	11 (10; 15)	0,963 (0,403; 2,888)	9,9 (9; 10)	5,349 (2,353; 10,56)
sanr200_0.7	18	1,885 (0,495; 2,75)	18	0,306 (0,095; 0,625)	15,8 (15; 18)	0,076 (0,02; 0,163)
sanr200_0.9	42	1,746 (1,291; 2,105)	41	0,266 (0,183; 0,379)	40,2 (38; 41)	0,159 (0,103; 0,232)
sanr400_0.5	13	6,094 (0,641; 8,646)	12,7 (12; 13)	1,506 (0,218; 3,54)	11,1 (11; 12)	0,521 (0,197; 0,96)
sanr400_0.7	21	5,005 (4,357; 5,5)	20,6 (20; 21)	1,593 (0,198; 4,331)	18 (17; 19)	0,563 (0,167; 1,055)

7.2 Kakovost algoritma simulirano ohlajanje

V tabeli 7.3 vidimo primerjavo med zaporedno različico simuliranega ohlajanja, s katero smo se seznanili v pričujočem delu, označimo jo z SA_C, in algoritmom simuliranega ohlajanja, opisanim v [14], objavljenim leta 2007, označimo ga z SA_GXXP. Preizkus je bil izveden na vseh 80 DIMACS primerkih. V tabeli so v drugem, tretjem in četrtem stolpcu navedene lastnosti primerkov, |V| označuje število vozlišč v grafu, |E| označuje število povezav in gostota označuje gostoto grafa, izračunano po standardni formuli (glej enačbo 7.1). Preizkus algoritma SA_C je bil izveden na strežniku Dell 1950. Po do sedaj zbranih informacijah ni boljše implementacije algoritma simulirano ohlajanje za reševanje problema največjega polnega podgrafa (ali za reševanje katerega izmed ekvivalentnih problemov). Avtorji so v [14] objavili rezultate preizkusa algoritma, napisanega v programskem jeziku C++, na računalniku s procesorjem Intel Pentium III, 866 MHz. Pri tem so kot rezultat navedli najboljši najdeni rezultat preko določeno število zagonov (odvisno od primerka, od 20 do 200). Kot je razvidno iz tabele, je predlagani algoritem SA_C dosegel vsaj enako kakovost rešitve pri vseh primerkih, razen pri hamming8-2 in hamming10-2. Boljšo kakovost rešitev je algoritem SA_C dosegel pri 35 primerkih.

$$\text{Gostota} = \frac{2|E|}{|V|(|V|-1)} \tag{7.1}$$

Čase za reševanje težje primerjamo zaradi različnih sistemov in ker avtorji niso navedli časov izvajanja standardnega primerjalnega preizkusa DIMACS. Kljub temu opazimo, da je predlagani algoritem SA_C pri določenih primerkih (npr. DCJC1000.5) za dva reda velikosti boljši od algoritma SA_GXXP pri enako dobrem najboljšem rezultatu.

Če primerjamo rešitve predlaganega algoritma SA_C z zelo uspešnim algoritmom dinamično lokalno iskanje, angl. *dynamic local search*, za reševanje problema največjega polnega podgrafa, leta 2006 opisanim v [43], ugotovimo, da je algoritem SA_C pri 67 DIMACS primerkih dosegel enako dobro rešitev kot tisti v [43] (v tabeli 7.3 označeno z zvezdico (*)). Pri nobenem izmed primerkov algoritem SA_C ni dosegel boljše kakovosti rešitve.

Tabela 7.3: Primerjava zaporednega algoritma simulirano ohlajanje z obstoječim algoritmom za problem največje neodvisne množice (Dell 1950)

Primerek	|V|	|E|	Gostota	SA_GXXP [14]		Predlagani SA_C	
				Kakovost	Čas (s)	Kakovost	Čas (s)
brock200_1	200	14834	0,745	21	5	21*	2,608 (0,659; 3,367)
brock200_2	200	9876	0,496	12	8	12*	0,128 (0,032; 0,281)
brock200_3	200	12048	0,605	14	2	15*	0,141 (0,029; 0,321)
brock200_4	200	13089	0,658	16	<1	17*	1,368 (0,448; 2,585)
brock400_1	400	59723	0,748	25	22	25	5,2 (4,044; 5,766)
brock400_2	400	59786	0,749	25	29	25,3 (25; 28)	5,32 (3,886; 6,098)
brock400_3	400	59681	0,748	25	26	26,3 (25; 30)	4,296 (2,752; 5,465)
brock400_4	400	59765	0,749	25	26	29,1 (25; 33*)	5,384 (1,458; 6,407)
brock800_1	800	207505	0,649	21	131	20,4 (20; 21)	11,627 (1,483; 15,63)
brock800_2	800	208166	0,651	21	124	20,7 (20; 21)	13,582 (10,92; 16,951)
brock800_3	800	207333	0,649	21	122	21,2 (20; 22)	13,209 (11,083; 14,475)
brock800_4	800	207643	0,650	21	125	20,6 (20; 21)	13,137 (9,318; 15,23)
C125.9	125	6963	0,898	34	<1	34*	0,687 (0,332; 0,97)
C250.9	250	27984	0,899	44	4	44*	1,879 (1,64; 2,065)
C500.9	500	112332	0,900	57	59	56,2 (56; 57*)	4,093 (3,443; 5,128)
C1000.9	1000	450079	0,901	68	222	66,8 (65; 68*)	7,676 (6,154; 10,535)
C2000.5	2000	999836	0,500	16	877	15,9 (15; 16*)	40,907 (3,341; 52,638)
C2000.9	2000	1799532	0,900	74	776	74,9 (74; 76)	17,221 (14,204; 21,053)
C4000.5	4000	4000268	0,500	17	903	16,7 (16; 17)	67,926 (12,73; 111,705)
c-fat200-1	200	1534	0,077	12	<1	12*	0,034 (0,024; 0,059)
c-fat200-2	200	3235	0,163	24	<1	22,2 (22; 24*)	0,031 (0,023; 0,055)
c-fat200-5	200	8473	0,426	58	<1	56,8 (56; 58*)	0,027 (0,016; 0,042)
c-fat500-1	500	4459	0,036	14	4	14*	0,216 (0,165; 0,405)
c-fat500-2	500	9139	0,073	26	<1	25,2 (24; 26*)	0,205 (0,158; 0,388)
c-fat500-5	500	23191	0,186	64	<1	62,4 (62; 64*)	0,183 (0,141; 0,349)
c-fat500-10	500	46627	0,374	126	<1	125,3 (124; 126*)	0,16 (0,112; 0,465)
DSJC500.5	500	62624	0,502	13	17	13*	4,121 (0,525; 8,359)
DSJC1000.5	1000	249826	0,500	15	363	14,5 (14; 15*)	13,638 (1,342; 26,532)
gen200_p0.9_44	200	17910	0,900	44	21	44*	1,637 (1,409; 1,999)
gen200_p0.9_55	200	17910	0,900	55	1	55*	1,637 (1,209; 1,862)
gen400_p0.9_55	400	71820	0,900	55	31	55*	3,304 (2,773; 3,923)
gen400_p0.9_65	400	71820	0,900	65	28	65*	3,434 (2,493; 4,513)
gen400_p0.9_75	400	71820	0,900	75	75	75*	3,871 (3,689; 4,159)
hamming6-2	64	1824	0,905	32	<1	31,2 (31; 32*)	0,674 (0,001; 1,207)
hamming6-4	64	704	0,349	4	<1	4*	0,007 (0,003; 0,014)
hamming8-2	256	31616	0,969	128	3	127	2,022 (1,991; 2,076)
hamming8-4	256	20864	0,639	16	<1	16*	0,082 (0,023; 0,178)
hamming10-2	1024	518656	0,990	512	427	511	5,641 (5,216; 6,205)
hamming10-4	1024	434176	0,829	40	144	40*	10,374 (10; 10,864)
johnson8-2-4	28	210	0,556	4	<1	4*	0,007 (0,001; 0,028)
johnson8-4-4	70	1855	0,768	14	<1	14*	0,006 (0,002; 0,01)
johnson16-2-4	120	5460	0,765	8	<1	8*	0,008 (0,003; 0,017)
johnson32-2-4	496	107880	0,879	16	<1	16*	0,035 (0,022; 0,077)
keller4	171	9435	0,649	11	<1	11*	0,028 (0,009; 0,093)
keller5	776	225990	0,752	27	143	27*	8,154 (7,313; 8,794)
keller6	3361	4619898	0,818	51	644	57,4 (55; 59*)	33,437 (23,748; 41,559)
MANN_a9	45	918	0,927	16	<1	16*	0,006 (0,001; 0,022)
MANN_a27	378	70551	0,990	126	49	125,8 (125; 126*)	4,097 (3,154; 5,332)
MANN_a45	1035	533115	0,996	334	393	338,9 (338; 340)	11,159 (10,299; 11,985)
MANN_a81	3321	5506380	0,999	1080	1879	1082,8 (1081; 1085)	0,084 (0,055; 0,265)
p_hat300-1	300	10933	0,244	8	<1	8*	0,146 (0,072; 0,339)
p_hat300-2	300	21928	0,489	25	<1	25*	1,865 (0,242; 2,898)
p_hat300-3	300	33390	0,744	36	2	36*	2,844 (2,345; 3,41)

(Se nadaljuje)

Tabela 7.3: Primerjava zaporednega algoritma simulirano ohlajanje z obstoječim algoritmom za problem največje neodvisne množice (Dell 1950, nadaljevanje)

Primerek	|V|	|E|	Gostota	SA_GXXP [14]		Predlagani SA_C	
				Kakovost	Čas (s)	Kakovost	Čas (s)
p_hat500-1	500	31569	0,253	9	<1	9*	0,271 (0,141; 0,583)
p_hat500-2	500	62946	0,505	36	1	36*	4,319 (2,649; 5,059)
p_hat500-3	500	93800	0,752	50	32	50*	4,198 (3,757; 4,703)
p_hat700-1	700	60999	0,249	11	18	11*	8,425 (1,645; 17,681)
p_hat700-2	700	121728	0,498	44	3	44*	6,353 (5,251; 7,027)
p_hat700-3	700	183010	0,748	62	12	62*	4,452 (3,932; 5,25)
p_hat1000-1	1000	122253	0,245	10	6	10*	2,376 (0,832; 5,9)
p_hat1000-2	1000	244799	0,490	46	16	46*	9,536 (7,977; 11,394)
p_hat1000-3	1000	371746	0,744	68	100	68*	7,877 (6,135; 8,687)
p_hat1500-1	1500	284923	0,253	12	490	11,8 (11; 12*)	36,515 (1,478; 52,699)
p_hat1500-2	1500	568960	0,506	65	40	65*	13,701 (11,894; 14,951)
p_hat1500-3	1500	847244	0,754	94	215	94*	10,444 (9,235; 11,855)
san200_0.7_1	200	13930	0,700	17	9	29 (21; 30*)	3,75 (0,085; 5,295)
san200_0.7_2	200	13930	0,700	15	9	16,7 (15; 18*)	2,714 (0,026; 4,612)
san200_0.9_1	200	17910	0,900	61	12	69,6 (69; 70*)	2,402 (1,96; 2,982)
san200_0.9_2	200	17910	0,900	60	12	60*	2,391 (2,235; 2,508)
san200_0.9_3	200	17910	0,900	44	6	44*	1,836 (1,634; 1,981)
san400_0.5_1	400	39900	0,500	7	<1	13*	1,313 (0,166; 3,094)
san400_0.7_1	400	55860	0,700	21	36	23,6 (21; 40*)	5,401 (0,034; 9,271)
san400_0.7_2	400	55860	0,700	16	25	20,2 (19; 30*)	7,945 (0,531; 10,033)
san400_0.7_3	400	55860	0,700	17	30	18,9 (17; 22*)	7,351 (0,339; 8,972)
san400_0.9_1	400	71820	0,900	57	38	100*	4,652 (4,005; 5,602)
san1000	1000	250500	0,502	8	<1	11,3 (10; 15*)	3,693 (0,385; 16,656)
sanr200_0.7	200	13868	0,697	18	<1	18*	1,885 (0,495; 2,75)
sanr200_0.9	200	17863	0,898	42	5	42*	1,746 (1,291; 2,105)
sanr400_0.5	400	39984	0,501	13	17	13*	6,094 (0,641; 8,646)
sanr400_0.7	400	55869	0,700	21	13	21*	5,005 (4,357; 5,5)

7.3 Cena komunikacije

Tabela 7.5 predstavlja primerjavo časov izvajanja treh metahevrističnih algoritmov med OpenMPI in PVM na LSC Adria. Edina razlika med OpemMPI in PVM zagoni je v tem, da se uporablja drugo orodje, sicer pa je potek algoritmov enak (enako število iteracij, enaka kakovost rešitve, enak vzorec komunikacije med procesi). Preizkus je bil izveden z uporabo 16 procesorjev na superračunalniku LSC Adria, pri čemer je na vsakem procesorju tekel natanko en proces.

Čase izvajanja lahko statistično analiziramo z uporabo Studentovega t-preizkusa značilnosti (parametrični in parni). Pri tem postavimo ničelno hipotezo, da sta populaciji časov izvajanja pri OpenMPI in PVM enaki. Za stopnjo zaupanja vzamemo 95%, kar pomeni, da ničelno hipotezo zavrnemo, kadar je vrednost p manjša kot 0,05. V tabeli 7.4 vidimo rezultate preizkusa. Ugotovimo lahko, da je pri simuliranem ohlajanju in razpršenem iskanju razlika vedno statistično pomembna. S pomočjo tabele 7.5 ugotovimo, da je povprečni čas izvajanja pri algoritmih simulirano ohlajanje in razpršeno iskanje pri PVM manjši od povprečnega časa izvajanja pri OpenMPI, kar pomeni, da je komunikacija med procesi s pomočjo PVM učinkovitejša od OpenMPI pri teh dveh algoritmih. Če pogledamo še navzkrižno entropijo, lahko ugotovimo, da je razlika statistično pomembna pri vseh primerkih, razen C4000.5 in keller6, za slednjega ni podatka. Ko upoštevamo še čase v tabeli 7.5, ugotovimo, da je OpenMPI boljši pri primerkih brock200_1 in MANN_a45 ter da je PVM boljši pri primerkih C1000.9 in MANN_a81. Torej za navzkrižno entropijo ne moremo reči, kateri algoritem je boljši, saj se rezultat nagiba bodisi k OpenMPI bodisi k PVM, odvisno od primerka.

Vendar moramo pri navzkrižni entropiji upoštevati še to, da je komunikacija pri PVM bolj zahtevna kot pri OpenMPI, ker PVM ne podpira operacije All_reduce in je bila nadomeščena z zaporedjem redukcije in razpošiljanja. Vendar to ni najboljše možno nadomestilo za operacijo All_reduce. Kolikor bi pri PVM implementacijo te operacije izboljšali, bi se morda izkazalo, da je uporaba PVM učinkovitejša.

V tabeli 7.6 vidimo enako primerjavo, kot je opisana v prvem odstavku, izvedeno na strežniku Dell R200. Preizkus je bil izveden z uporabo enega procesorja s štirimi jedri, pri čemer je na vsakem jedru tekel natanko en proces. Rezultati te tabele se bistveno razlikujejo od preizkusa na LSC Adria. Pri simuliranem ohlajanju in razpršenem iskanju vidimo, da se razmerje med časom OpenMPI in PVM giblje zelo blizu 1 (razen za primerek MANN_a9). Anomalijo pri primerku MANN_a9 lahko pripišemo temu, da je čas izvajanja

Tabela 7.4: Studentov t-preizkus časa izvajanja vzporednega algoritma med OpenMPI in PVM za problem največje neodvisne množice (LSC Adria)

(a) Simulirano ohlajanje

Primerek	p-vrednost	Interval zaupanja	
brock200_1	1.11E-07	6.430	8.708
C1000.9	8.29E-09	15.333	19.199
C4000.5	3.75E-09	55.273	67.876
keller6	1.75E-10	39.658	45.857
MANN_a9	9.08E-07	4.733	6.986
MANN_a45	6.58E-10	14.930	17.672
MANN_a81	0.005	3.189	13.309

(b) Razpršeno iskanje

Primerek	p-vrednost	Interval zaupanja	
brock200_1	1.05E-08	5.893	7.423
C1000.9	2.01E-14	9.077	9.568
C4000.5	7.60E-06	11.811	19.593
keller6	1.05E-12	17.537	19.035
MANN_a9	2.87E-12	2.006	2.199
MANN_a45	4.88E-10	6.906	8.129
MANN_a81	0.003	2.323	8.608

(c) Navzkrižna entropija

Primerek	p-vrednost	Interval zaupanja	
brock200_1	3.36E-17	3.825	3.925
C1000.9	1.43E-09	-22.541	-18.751
C4000.5	0.648	-6.390	9.760
keller6	*N.P.*	*N.P.*	*N.P.*
MANN_a9	9.99E-08	2.025	2.733
MANN_a45	0.002	0.814	2.537
MANN_a81	0.015	-71.818	-11.223

za ta primerek izredno majhen in se zato bolj odraža cena postavitve okolja OpenMPI (PVM te cene nima, saj je okolje že postavljeno). Za razliko od LSC Adrie pri tem preizkusu ne bomo mogli ugotoviti, da je razlika med OpenMPI in PVM statistično pomembna, saj so si vrednosti preveč podobne. Če pogledamo še navzkrižno entropijo, opazimo, da je OpenMPI hitrejši pri primerkih brock200_1 in MANN_a9. To sta primerka, katerih čas izvajanja je majhen in zato potrjuje dejstvo, da je potrebno pri OpenMPI najprej postaviti okolje, pri PVM pa ne. Pri ostalih primerkih pa opazimo, da je OpenMPI hitrejši za približno 2- do 5-krat. Kot že omenjeno v prejšnjem odstavku, izvajata OpenMPI in PVM algoritma operacijo All_reduce na različen način, kar ima za posledico pričujoče rezultate.

Analizo zaključimo z ugotovitvijo, da se algoritem, ki uporablja OpenMPI, izvaja približno enako dolgo kot algoritem, ki uporablja PVM, pod pogojem, da tečejo vsi procesi na istem procesorju (in morda tudi računalniku). To namiguje, da je razlika v rezultatih na LSC Adria posledica različno hitrega odziva in prenosa podatkov preko omrežja, saj OpenMPI in PVM, kot že omenjeno, uporabljata različne načine komunikacije. Takšno ugotovitev bi lahko dokončno potrdili, če bi izvedli podobne preizkuse še na LSC Adria.

Tabela 7.5: Primerjava časa izvajanja vzporednega algoritma med OpenMPI in PVM za problem največje neodvisne množice (LSC Adria)

Simulirano ohlajanje Primerek	Število iteracij	Kakovost	Čas OpenMPI	Čas PVM	Razmerje čas OpenMPI/PVM
brock200_1	4 (2; 8)	21	8,967 (5,72; 12,326)	1,398 (0,629; 2,704)	6,414
C1000.5	9,4 (8; 11)	67,8 (67; 68)	26,127 (18,972; 31,14)	8,861 (7,615; 10,267)	2,949
C4000.5	7,8 (2; 11)	17	123,555 (67,822; 149,694)	61,981 (19,67; 79,397)	1,993
keller6	8,7 (8; 9)	59	74,895 (65,963; 82,089)	32,138 (30,183; 33,545)	2,33
MANN_a9	1,9 (1; 2)	16	5,881 (2,071; 7,325)	0,022 (0,019; 0,028)	267,318
MANN_a45	17,3 (15; 18)	340	31,4 (28,42; 34,416)	15,1 (13,559; 15,571)	2,079
MANN_a81	3,2 (1; 13)	1082,3 (1081; 1088)	15,125 (2,217; 56,212)	6,876 (0,171; 36,532)	2,2
Razpršeno iskanje Primerek	Število iteracij	Kakovost	Čas OpenMPI	Čas PVM	Razmerje čas OpenMPI/PVM
brock200_1	2,1 (1; 4)	21	7,146 (5,406; 8,18)	0,488 (0,299; 0,891)	14,643
C1000.5	6,8 (4; 15)	62,1 (60; 64)	15,821 (13,094; 24,297)	6,499 (4,457; 14,222)	2,434
C4000.5	2,3 (1; 5)	16	31,476 (8,522; 46,911)	15,774 (6,393; 27,788)	1,995
keller6	6,9 (4; 13)	53,8 (52; 56)	42,63 (34,679; 62,355)	24,344 (13,833; 43,86)	1,751
MANN_a9	1	16	2,114 (2,066; 2,494)	0,012 (0,008; 0,034)	176,167
MANN_a45	3,3 (1; 5)	339,1 (338; 340)	10,648 (6,411; 12,925)	3,131 (1,323; 5,136)	3,401
MANN_a81	1	1088,3 (1088; 1089)	7,32 (2,29; 14,923)	1,854 (0,229; 4,311)	3,948
Navzkrižna entropija Primerek	Število iteracij	Kakovost	Čas OpenMPI	Čas PVM	Razmerje čas OpenMPI/PVM
brock200_1	7,4 (5; 11)	20,1 (20; 21)	4,137 (4,129; 4,15)	0,263 (0,163; 0,379)	15,73
C1000.5	26,8 (23; 32)	64,1 (64; 65)	14,94 (13,39; 17,201)	35,586 (29,675; 43,105)	0,42
C4000.5	1	15,2 (15; 16)	43,71 (26,031; 50,871)	42,025 (39,083; 45,534)	1,04
keller6	33,2 (5; 42)	49,9 (48; 51)	525,216 (360,896; 630,092)	*N.P.*	*N.P.*
MANN_a9	1	16	2,39 (2,07; 3,128)	0,011 (0,009; 0,014)	217,273
MANN_a45	3 (1; 6)	338	5,587 (3,618; 7,515)	3,912 (0,819; 8,417)	1,428
MANN_a81	6 (1; 9)	1090	137,854 (39,378; 197,634)	179,374 (27,859; 271,149)	0,769

Tabela 7.6: Primerjava časa izvajanja vzporednega algoritma med OpenMPI in PVM za problem največje neodvisne množice (Dell R200)

Simulirano ohlajanje	Število	Kakovost	Čas	Čas	Razmerje čas
Primerek	iteracij		OpenMPI	PVM	OpenMPI/PVM
brock200_1	7,4 (2; 11)	21	2,161 (0,633; 2,763)	2,086 (0,586; 2,784)	1,036
C1000.9	7,7 (7; 9)	66,8 (66; 68)	6,069 (5,067; 8,066)	6,069 (5,282; 7,461)	1
C4000.5	2	17	57,705 (19,064; 75,942)	57,637 (19,14; 75,768)	1,001
keller6	7,6 (7; 9)	58 (57; 59)	23,968 (19,896; 27,282)	24,135 (19,82; 28,076)	0,993
MANN_a9	1,9 (1; 2)	16	0,231 (0,057; 0,261)	0,022 (0,009; 0,03)	10,5
MANN_a45	16,7 (16; 19)	339,7 (339; 340)	8,707 (8,182; 9,811)	8,935 (8,117; 9,527)	0,974
MANN_a81	1	1081,9 (1080; 1088)	1,06 (0,107; 6,401)	1,047 (0,06; 6,526)	1,012
Razpršeno iskanje	Število	Kakovost	Čas	Čas	Razmerje čas
Primerek	iteracij		OpenMPI	PVM	OpenMPI/PVM
brock200_1	2,7 (1; 5)	21	0,336 (0,221; 0,559)	0,326 (0,201; 0,54)	1,031
C1000.9	6,1 (5; 8)	61,1 (59; 63)	2,621 (2,171; 3,138)	2,646 (2,05; 3,135)	0,991
C4000.5	4,5 (1; 11)	16	13,513 (6,808; 24,573)	13,504 (6,452; 24,667)	1,001
keller6	7,1 (5; 13)	51,8 (49; 54)	11,98 (8,551; 19,838)	11,984 (8,64; 20,338)	1
MANN_a9	1	16	0,068 (0,055; 0,078)	0,013 (0,011; 0,029)	5,231
MANN_a45	4,3 (1; 13)	338,4 (338; 339)	1,751 (0,098; 4,699)	1,798 (0,042; 4,83)	0,974
MANN_a81	1,5 (1; 6)	1088,3 (1088; 1089)	1,444 (0,188; 6,945)	1,454 (0,144; 7,221)	0,993
Navzkrižna entropija	Število	Kakovost	Čas	Čas	Razmerje čas
Primerek	iteracij		OpenMPI	PVM	OpenMPI/PVM
brock200_1	8,1 (3; 12)	19,6 (19; 20)	4,137 (4,129; 4,15)	0,824 (0,401; 1,126)	5,021
C1000.9	31 (28; 35)	63,4 (62; 64)	14,94 (13,39; 17,201)	123,405 (113,771; 137,94)	0,121
C4000.5	1,3 (1; 2)	15	43,71 (26,031; 50,871)	167,205 (119,087; 278,231)	0,261
keller6	10,5 (1; 42)	46,9 (46; 50)	525,216 (360,896; 630,092)	1033,527 (71,009; 3665,111)	0,508
MANN_a9	1,1 (1; 2)	16	2,39 (2,07; 3,128)	0,022 (0,01; 0,07)	108,636
MANN_a45	4,8 (2; 9)	338	5,587 (3,618; 7,515)	20,394 (7,839; 39,899)	0,274
MANN_a81	6,2 (2; 11)	1089 (1088; 1090)	137,854 (39,378; 197,634)	689,687 (147,133; 1069,154)	0,2

7.4 Čas izvajanja in število iteracij

Tabela 7.7 prikazuje primerjavo med zaporednimi in vzporednimi metahevrističnimi algoritmi glede na število iteracij in čas pri fiksni kakovosti rešitve. Preizkus je bil izveden na superračunalniku LSC Adria z uporabo 16 procesov na enak način kot v prejšnjem razdelku. Če želimo primerjati čas izvajanja in število iteracij, moramo opraviti meritve pri istem rezultatu, zato fiksiramo kakovosti rešitev. Primerjava je mogoča le, če fiksiramo kakovost rešitve na najmanjšo doseženo kakovost rešitve pri določenem primerku, vzeto čez zaporedni in vzporedni algoritem.

Ugotovimo lahko, da je pri simuliranem ohlajanju povprečno število iteracij pri vseh primerkih, razen pri MANN_a81, manjše kot pri zaporednem algoritmu za razmerje med 1 in 3. Pri razpršenem iskanju in pri navzkrižni entropiji je vzporedni algoritem pri vseh sedmih primerkih vsaj tako dober kot zaporedni algoritem, razmerje se giblje med 1 in 4.

Ker imamo meritve tako zaporednega kot vzporednega algoritma, lahko izračunamo pospešitev. Pri preizkusu je bilo uporabljenih 16 procesov, zato je največja možna pospešitev 16. Kot vidimo v zadnjih dveh stolpcih, je bila pospešitev bistveno manjša od 16. Pri PVM je skoraj pri vseh primerkih za vse tri algoritme nad 1. Drugače je bilo pri OpenMPI, kjer je bila pospešitev manjša od 1 za vse primerke pri simuliranem ohlajanju in navzkrižni entropiji. Kadar je pospešitev manjša od 1, je bolje, da izvajamo zaporedni algoritem kot vzporednega.

Iz tabele 7.8 vidimo enako primerjavo, kot je opisana v prejšnjem odstavku, izvedeno na strežniku Dell R200. Preizkus je bil izveden z uporabo štirih procesorjev, pri čemer je na vsakem procesorju tekel natanko en proces. Vidimo, da je pri simuliranem ohlajanju razmerje iteracij precej blizu 1. Pri preizkusu na LSC Adria ni bilo tako, ker je bilo tam uporabljenih 16 procesov, in ne samo štirje. Največja možna pospešitev je 4, vendar kot je razvidno iz zadnjih dveh stolpcev, je pospešitev le malo večja od 1, razen za primerek MANN_a81. Če pogledamo še algoritma razpršeno iskanje in navzkrižna entropija, vidimo, da razmerje iteracij niha med 1 in 3, odvisno od primerka, kar je bolje od simuliranega ohlajanja. Pospešitve so pri razpršenem iskanju in navzkrižni entropiji sicer boljše kot pri simuliranem ohlajanju, vendar v večini še vedno precej manjše od 4. Ena izmed dobrih izjem je reševanje primerka C4000.5 z navzkrižno entropijo, kjer je pospešitev pri PVM enaka 2,685.

Zaključimo lahko, da preizkušeni vzporedni algoritmi niso dosegli dobre pospešitve. Eden izmed bolj pomembnih vzrokov je relativno majhno razmerje števila iteracij med zaporednimi in vzporednimi algoritmi, ki onemogoča, da bi pospešitev dobro naraščala s številom procesov. Za dobro pospešitev bi moralo biti število iteracij pri vzporednem algoritmu približno p-krat manjše kot pri zaporednem algoritmu.

Tabela 7.7: Primerjava časa izvajanja in števila iteracij med zaporednimi in vzporednimi algoritmi pri fiksni kakovosti rešitve za problem največje neodvisne množice (LSC Adria)

Simulirano ohlajanje		Zaporedni algoritem		Vzporedni algoritem					Razmerje iteracij
Primerek	Kakovost	Čas	Število iteracij	Čas OpenMPI	Čas PVM	Število iteracij	Pospešitev OpenMPI	Pospešitev PVM	zaporedni / vzporedni
brock200_1	21	2,6 (1; 3)	8,2 (2; 11)	8 (5; 11)	1,4 ($< \epsilon$; 3)	3 (1; 7)	0,325	1,857	2,733
C1000.5	65	9,6 (8; 11)	8,6 (7; 10)	22,2 (19; 25)	7,1 (6; 8)	6,3 (5; 7)	0,432	1,352	1,365
C4000.5	16	27,3 (20; 40)	2,7 (2; 4)	65,8 (65; 66)	19,5 (19; 20)	1	0,415	1,4	2,7
keller6	55	30,2 (23; 36)	8,1 (6; 10)	63,4 (57; 71)	25,4 (23; 27)	5,6 (5; 6)	0,476	1,189	1,446
MANN_a9	16	$< \epsilon$	1,9 (1; 2)	4,9 (2; 6)	$< \epsilon$	1	*N.M.I*	*N.M.I*	1,9
MANN_a45	338	16,2 (15; 17)	18,2 (17; 19)	25,5 (21; 28)	11,9 (11; 13)	12,1 (11; 14)	0,635	1,361	1,504
MANN_a81	1081	$< \epsilon$	1	12,1 (2; 30)	4,9 ($< \epsilon$; 16)	2 (1; 5)	*N.M.I*	*N.M.I*	0,5
Razpršeno iskanje		Zaporedni algoritem		Vzporedni algoritem					Razmerje iteracij
Primerek	Kakovost	Čas	Število iteracij	Čas OpenMPI	Čas PVM	Število iteracij	Pospešitev OpenMPI	Pospešitev PVM	zaporedni / vzporedni
brock200_1	20	1	2,6 (2; 3)	5,3 (5; 6)	0,1 ($< \epsilon$; 1)	1	0,189	10	2,6
C1000.5	58	5,1 (3; 9)	8 (5; 14)	12,4 (10; 14)	3,3 (2; 4)	3,3 (2; 4)	0,411	1,545	2,424
C4000.5	15	6 (5; 9)	1,7 (1; 3)	22,8 (22; 23)	8,7 (8; 9)	1	0,263	0,69	1,7
keller6	48	18,9 (12; 51)	8,5 (6; 22)	27,2 (26; 30)	9,4 (8; 12)	2,3 (2; 3)	0,695	2,011	3,696
MANN_a9	16	0,2 ($< \epsilon$; 1)	1	5,1 (5; 6)	$< \epsilon$	1	0,039	*N.M.I*	1
MANN_a45	337	1,2 ($< \epsilon$; 6)	2,8 (1; 10)	6,1 (6; 7)	1	1	0,197	1,2	2,8
MANN_a81	1087	4,2 (0; 41)	2,9 (1; 20)	13,8 (13; 14)	3,1 (3; 4)	1	0,304	1,355	2,9
Navzkrižna entropija		Zaporedni algoritem		Vzporedni algoritem					Razmerje iteracij
Primerek	Kakovost	Čas	Število iteracij	Čas OpenMPI	Čas PVM	Število iteracij	Pospešitev OpenMPI	Pospešitev PVM	zaporedni / vzporedni
brock200_1	18	$< \epsilon$	4,1 (1; 9)	2,4 (2; 4)	$< \epsilon$	1,2 (1; 2)	*N.M.I*	*N.M.I*	3,417
C1000.5	52	2,5 ($< \epsilon$; 5)	4,8 (1; 9)	2,5 (2; 3)	0,7 (0; 1)	1	1	3,571	4,8
C4000.5	15	129,9 (39; 369)	3,6 (1; 10)	30,1 (26; 31)	42,7 (38; 45)	1	4,316	3,042	3,6
keller6	43	33,4 (20; 65)	1,5 (1; 3)	17,1 (15; 18)	*N.P.*	1	1,953	*N.P.*	1,5
MANN_a9	16	$< \epsilon$	1,4 (1; 3)	2	$< \epsilon$	1	*N.M.I*	*N.M.I*	1,4
MANN_a45	336	1,1 (1; 2)	1,2 (1; 3)	2,6 (2; 3)	0,8 ($< \epsilon$; 1)	1	0,423	1,375	1,2
MANN_a81	1088	91 (26; 208)	3,429 (1; 8)	26,571 (21; 58)	30,714 (26; 49)	1,143 (1; 2)	3,425	2,963	3

Tabela 7.8: Primerjava časa izvajanja in števila iteracij med zaporednimi in vzporednimi algoritmi pri fiksni kakovosti rešitve za problem največje neodvisne množice (Dell R200)

Simulirano ohlajanje		Zaporedni algoritem		Vzporedni algoritem					Razmerje iteracij
Primerek	Kakovost	Čas	Število iteracij	Čas OpenMPI	Čas PVM	Število iteracij	Pospešitev OpenMPI	Pospešitev PVM	zaporedni / vzporedni
brock200_1	21	1,8 ($< \epsilon$; 3)	8,2 (2; 11)	2,1 (0; 3)	2 ($< \epsilon$; 3)	7,4 (2; 11)	0,857	0,9	1,108
C1000.9	65	5,5 (5; 6)	8,6 (7; 10)	4,9 (4; 7)	5,1 (4; 6)	7,7 (7; 9)	1,122	1,078	1,117
C4000.5	16	29,8 (19; 47)	2,7 (2; 4)	19,3 (19; 20)	19,2 (19; 20)	2	1,544	1,552	1,35
keller6	55	22,2 (18; 26)	8,1 (6; 10)	21,3 (20; 24)	21,3 (20; 24)	7,6 (7; 9)	1,042	1,042	1,066
MANN_a9	16	0,1 ($< \epsilon$; 1)	1,9 (1; 2)	0,3 ($< \epsilon$; 1)	$< \epsilon$	1,9 (1; 2)	0,333	N.M.I.	1
MANN_a45	338	8,5 (8; 9)	18,2 (17; 19)	7,7 (7; 9)	7,9 (7; 9)	16,7 (16; 19)	1,104	1,076	1,09
MANN_a81	1081	0,3 ($< \epsilon$; 1)	1	0,1 ($< \epsilon$; 1)	0,1 ($< \epsilon$; 1)	1	3	3	1

Razpršeno iskanje		Zaporedni algoritem		Vzporedni algoritem					Razmerje iteracij
Primerek	Kakovost	Čas	Število iteracij	Čas OpenMPI	Čas PVM	Število / iteracij	Pospešitev OpenMPI	Pospešitev PVM	zaporedni / vzporedni
brock200_1	20	0,4 ($< \epsilon$; 1)	2,6 (2; 3)	$< \epsilon$	0,3 ($< \epsilon$; 1)	1	N.M.I.	1,333	2,6
C1000.9	58	2,8 (1; 5)	8 (5; 14)	2 (1; 3)	2,1 (2; 3)	4,1 (3; 5)	1,4	1,333	1,951
C4000.5	15	5,9 (5; 8)	1,7 (1; 3)	7 (6; 8)	6,9 (6; 7)	1	0,843	0,855	1,7
keller6	48	11,6 (8; 27)	8,5 (6; 22)	6,8 (5; 9)	7 (5; 9)	3,4 (2; 5)	1,706	1,657	2,5
MANN_a9	16	$< \epsilon$	1	0,2 ($< \epsilon$; 1)	0,1 ($< \epsilon$; 1)	1	N.M.I.	N.M.I.	1
MANN_a45	337	0,5 ($< \epsilon$; 3)	2,8 (1; 10)	0,4 ($< \epsilon$; 1)	0,3 ($< \epsilon$; 1)	1	1,25	1,667	2,8
MANN_a81	1087	2,1 ($< \epsilon$; 21)	2,9 (1; 20)	1,4 (1; 2)	1,6 (1; 2)	1	1,5	1,313	2,9

Navzkrižna entropija		Zaporedni algoritem		Vzporedni algoritem					Razmerje iteracij
Primerek	Kakovost	Čas	Število iteracij	Čas OpenMPI	Čas PVM	Število iteracij	Pospešitev OpenMPI	Pospešitev PVM	zaporedni / vzporedni
brock200_1	18	$< \epsilon$	4,1 (1; 9)	0,2 ($< \epsilon$; 1)	$< \epsilon$	2,4 (1; 4)	N.M.I.	N.M.I.	1,708
C1000.9	52	9,1 (2; 18)	4,8 (1; 9)	4,6 (2; 11)	5,9 (2; 17)	2,1 (1; 5)	1,978	1,542	2,286
C4000.5	15	395,5 (87; 1156)	3,6 (1; 10)	156,5 (119; 242)	147,3 (119; 209)	1,3 (1; 2)	2,527	2,685	2,769
keller6	43	98,4 (56; 156)	1,5 (1; 3)	72,1 (71; 73)	71,8 (71; 75)	1	1,365	1,37	1,5
MANN_a9	16	$< \epsilon$	1,4 (1; 3)	0,1 ($< \epsilon$; 1)	$< \epsilon$	1,1 (1; 2)	N.M.I.	N.M.I.	1,273
MANN_a45	336	3,2 (2; 8)	1,2 (1; 3)	2,9 (2; 3)	2,6 (2; 3)	1	1,103	1,231	1,2
MANN_a81	1088	233,3 (79; 474)	3,2 (1; 8)	147,9 (81; 248)	147,4 (80; 259)	1,8 (1; 3)	1,577	1,583	1,778

7.5 Kakovost rešitev in število iteracij

V tabeli 7.9 vidimo primerjavo med zaporednimi in vzporednimi metahevrističnimi algoritmi glede na kakovost rešitve in število iteracij. Preizkus je bil izveden na LSC Adria z uporabo 16-ih procesov, pod že opisanimi pogoji. Pri preizkusu so imeli vsi algoritmi podano ciljno kakovost rešitve, kot je razvidno iz drugega stolpca tabele. Opazimo lahko, da je povprečna kakovost rešitev pri vseh treh vzporednih algoritmih enaka ali boljša od kakovosti zaporednega algoritma. Razlika je relativno majhna, med 0 in 10 odstotki. Razlog za to je, da je preiskanost prostora pri vzporednem algoritmu večja in da je vzporedni algoritem, potem ko je enkrat presegel prag želene kakovosti, le-tega presegel za več kot zaporedni algoritem, saj je imel hkrati na razpolago rezultate 16 procesov, in ne samo enega, kot je to pri zaporednem algoritmu.

Če pogledamo še razmerje iteracij (ki je izračunano kot razmerje med povprečno vrednostjo števila iteracij vzporednega algoritma proti povprečni vrednosti števila iteracij zaporednega algoritma), vidimo, da je vzporedni algoritem pri vseh treh metahevrističnih algoritmih v večini primerov potreboval manj iteracij kot zaporedni algoritem (razmerje manjše od 1). Vendar hkrati vidimo tudi to, da razmerje nikoli ni manjše od 0,2. V takšnem primeru je najboljša možna teoretična pospešitev 5, kar pomeni, da bi lahko algoritem učinkovito uporabil do 5 procesov. To slabo razmerje iteracij ima za posledico, da imajo vsi trije metahevristični algoritmi relativno majhno pospešitev, kar smo videli že v prejšnjem razdelku.

V tabeli 7.10 vidimo enako primerjavo, kot je opisana v prvem odstavku, izvedeno na strežniku Dell R200 z uporabo štirih procesov. Če začnemo s simuliranim ohlajanjem, vidimo, da je pri primerku C4000.5 bistvena razlika v povprečnem številu iteracij, 2 pri Dell R200 in 7,8 pri LSC Adria. Pri nekem drugem primerku, npr. brock200_1, je razlika ravno obratna in je povprečno število iteracij pri Dell R200 7,4 ter na LSC Adria le 4. Rezultati namigujejo na to, da je težko reči, ali je bolje uporabiti štiri procese ali 16 procesov z vidika manjšega števila iteracij. Če primerjamo še kakovosti rešitev, lahko ugotovimo, da rezultati pri uporabi 16 procesov nikoli niso slabši od tistih pri uporabi štirih procesov, kvečjemu boljši. Če analiziranje nadaljujemo na razpršenem iskanju, ugotovimo, da je število iteracij pri 16 procesih vedno manjše (razen pri primerku C1000.9). To je verjetno posledica dejstva, da je pri uporabi 16 procesov preiskanost prostora večja, saj pri iskanju sodeluje 4-krat več procesov. Pri tem opazimo, da povprečna kakovost pri 16 procesih ni nikoli slabša od povprečne kakovosti pri štirih procesih. Pri navzkrižni entropiji lahko ugotovimo, da povečevanje števila procesov ugodno vpliva na kakovost rešitve. To si lahko ponovno pojasnimo z večjo preiskanostjo prostora, saj je pri 16 procesih generiranih 4-krat več naključnih rešitev. Podobno ugotovitev zabeležimo tudi za povprečno število iteracij, saj je pri 16 procesih vedno manjše

ali enako (razen pri primerku keller6). Podobno kot v prejšnjem odstavku ugotovimo, da je vredno uporabljati vzporedni algoritem navzkrižne entropije le, kadar je razmerje iteracij manjše od 1. To pri štirih procesih drži le za primerka C4000.5 in MANN_a9.

Zaključimo lahko z ugotovitvijo, da v splošnem povečanje števila procesov pripomore k večji preiskanosti prostora in zato k boljši kakovosti rešitve. Glede iteracij pri simuliranem ohlajanju ne moremo zaključiti nič konkretnega. Pri razpršenem iskanju in navzkrižni entropiji lahko rečemo, da se v splošnem s povečevanjem števila procesov število potrebnih iteracij zmanjšuje.

Tabela 7.9: Primerjava kakovosti rešitev in števila iteracij med zaporednimi in vzporednimi algoritmi za problem največje neodvisne množice (LSC Adria)

Simulirano ohlajanje	Ciljna	Zaporedni algoritem		Vzporedni algoritem		Razmerje kakovosti	Razmerje iteracij
Primerek	kakovost	Kakovost	Št. iteracij	Kakovost	Št. iteracij	vzporedni / zaporedni	vzporedni / zaporedni
brock200_1	21	21	8,2 (2; 11)	21	4 (2; 8)	1	0,488
C1000.5	68	66,8 (65; 68)	10,4 (8; 14)	67,8 (67; 68)	9,4 (8; 11)	1,015	0,904
C4000.5	17	16,7 (16; 17)	7,6 (2; 13)	17	7,8 (2; 11)	1,018	1,026
keller6	59	57,4 (55; 59)	10,8 (6; 15)	59	8,7 (8; 9)	1,028	0,806
MANN_a9	16	16	1,9 (1; 2)	16	1,9 (1; 2)	1	1
MANN_a45	340	338,9 (338; 340)	18,8 (18; 19)	340	17,3 (15; 18)	1,003	0,92
MANN_a81	1085	1082,8 (1081; 1085)	1	1082,3 (1081; 1088)	3,2 (1; 13)	1	3,2
Razpršeno iskanje	Ciljna	Zaporedni algoritem		Vzporedni algoritem		Razmerje kakovosti	Razmerje iteracij
Primerek	kakovost	Kakovost	Št. iteracij	Kakovost	Št. iteracij	vzporedni / zaporedni	vzporedni / zaporedni
brock200_1	21	20,3 (20; 21)	4,7 (2; 22)	21	2,1 (1; 4)	1,034	0,447
C1000.5	64	61,7 (58; 64)	14,5 (9; 38)	62,1 (60; 64)	6,8 (4; 15)	1,006	0,469
C4000.5	16	15,6 (15; 16)	6,9 (1; 18)	16	2,3 (1; 5)	1,026	0,333
keller6	55	51,2 (48; 55)	14,4 (7; 26)	53,8 (52; 56)	6,9 (4; 13)	1,051	0,479
MANN_a9	16	16	1	16	1	1	1
MANN_a45	339	338,1 (337; 339)	6,2 (1; 13)	339,1 (338; 340)	3,3 (1; 5)	1,003	0,532
MANN_a81	1088	1087,3 (1087; 1088)	4,7 (1; 20)	1088,3 (1088; 1089)	1	1,001	0,213
Navzkrižna entropija	Ciljna	Zaporedni algoritem		Vzporedni algoritem		Razmerje kakovosti	Razmerje iteracij
Primerek	kakovost	Kakovost	Št. iteracij	Kakovost	Št. iteracij	vzporedni / zaporedni	vzporedni / zaporedni
brock200_1	20	18,3 (18; 20)	6 (1; 17)	20,1 (20; 21)	7,4 (5; 11)	1,098	1,233
C1000.5	64	59,6 (52; 64)	24,6 (1; 41)	64,1 (64; 65)	26,8 (23; 32)	1,076	1,089
C4000.5	15	15	3,6 (1; 10)	15,2 (15; 16)	1	1,013	0,278
keller6	50	45,7 (43; 50)	8,5 (1; 28)	49,9 (48; 51)	33,2 (5; 42)	1,092	3,906
MANN_a9	16	16	1,4 (1; 3)	16	1	1	0,714
MANN_a45	338	336,9 (336; 338)	4,7 (1; 11)	338	3 (1; 6)	1,003	0,638
MANN_a81	1090	1088,571 (1088; 1090)	5,571 (1; 9)	1090	6 (1; 9)	1,001	1,077

Tabela 7.10: Primerjava kakovosti rešitev in števila iteracij med zaporednimi in vzporednimi algoritmi za problem največje neodvisne množice (Dell R200)

Simulirano ohlajanje	Ciljna	Zaporedni algoritem		Vzporedni algoritem		Razmerje kakovosti	Razmerje iteracij
Primerek	kakovost	Kakovost	Št. iteracij	Kakovost	Št. iteracij	vzporedni / zaporedni	vzporedni / zaporedni
brock200_1	21	21	8,2 (2; 11)	21	7,4 (2; 11)	1	0,902
C1000.9	68	66,8 (65; 68)	10,4 (8; 14)	66,8 (66; 68)	7,7 (7; 9)	1	0,74
C4000.5	17	16,7 (16; 17)	7,6 (2; 13)	17	2	1,018	0,263
keller6	59	57,4 (55; 59)	10,8 (6; 15)	58 (57; 59)	7,6 (7; 9)	1,01	0,704
MANN_a9	16	16	1,9 (1; 2)	16	1,9 (1; 2)	1	1
MANN_a45	340	338,9 (338; 340)	18,8 (18; 19)	339,7 (339; 340)	16,7 (16; 19)	1,002	0,888
MANN_a81	1085	1082,8 (1081; 1085)	1	1081,9 (1080; 1088)	1	1	1
Razpršeno iskanje	Ciljna	Zaporedni algoritem		Vzporedni algoritem		Razmerje kakovosti	Razmerje iteracij
Primerek	kakovost	Kakovost	Št. iteracij	Kakovost	Št. iteracij	vzporedni / zaporedni	vzporedni / zaporedni
brock200_1	21	20,3 (20; 21)	4,7 (2; 22)	21	2,7 (1; 5)	1,034	0,574
C1000.9	64	61,7 (58; 64)	14,5 (9; 38)	61,1 (59; 63)	6,1 (5; 8)	0,99	0,421
C4000.5	16	15,6 (15; 16)	6,9 (1; 18)	16	4,5 (1; 11)	1,026	0,652
keller6	55	51,2 (48; 55)	14,4 (7; 26)	51,8 (49; 54)	7,1 (5; 13)	1,012	0,493
MANN_a9	16	16	1	16	1	1	1
MANN_a45	339	338,1 (337; 339)	6,2 (1; 13)	338,4 (338; 339)	4,3 (1; 13)	1,001	0,694
MANN_a81	1088	1087,3 (1087; 1088)	4,7 (1; 20)	1088,3 (1088; 1089)	1,5 (1; 6)	1,001	0,319
Navzkrižna entropija	Ciljna	Zaporedni algoritem		Vzporedni algoritem		Razmerje kakovosti	Razmerje iteracij
Primerek	kakovost	Kakovost	Št. iteracij	Kakovost	Št. iteracij	vzporedni / zaporedni	vzporedni / zaporedni
brock200_1	20	18,3 (18; 20)	6 (1; 17)	19,6 (19; 20)	8,1 (3; 12)	1,071	1,35
C1000.9	64	59,6 (52; 64)	24,6 (1; 41)	63,4 (62; 64)	31 (28; 35)	1,064	1,26
C4000.5	15	15	3,6 (1; 10)	15	1,3 (1; 2)	1	0,361
keller6	50	45,7 (43; 50)	8,5 (1; 28)	46,9 (46; 50)	10,5 (1; 42)	1,026	1,235
MANN_a9	16	16	1,4 (1; 3)	16	1,1 (1; 2)	1	0,786
MANN_a45	338	336,9 (336; 338)	4,7 (1; 11)	338	4,8 (2; 9)	1,003	1,021
MANN_a81	1090	1088,6 (1088; 1090)	4,7 (1; 9)	1089 (1088; 1090)	6,2 (2; 11)	1	1,319

7.6 Profili kakovosti rešitev skozi iteracije

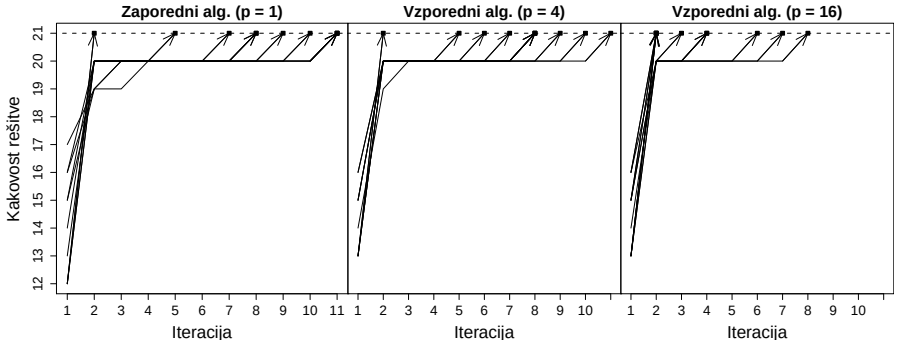
Na slikah 7.1, 7.2 in 7.3 vidimo primerjavo med tremi metahevrističnimi algoritmi glede na trenutno kakovost rešitve skozi iteracije. Za vsakega izmed algoritmov je bil narejen profil zaporednega zagona in dva profila vzporednega zagona (en s štirimi procesi in en s 16 procesi). Na vodoravni osi so prikazane iteracije in na navpični osi kakovosti rešitev. Končna rešitev je označena s polnim kvadratkom, v katerega vodi puščica. Prikazani so profili vseh desetih zagonov, razen pri navzkrižni entropiji, primerek MANN_a81, za katerega je prikazanih le sedem zagonov. Kot je zapisano v oznakah slik, so imeli vsi zagoni podano ciljno kakovost (enako kot v tabelah 7.9 in 7.10). Pri vsakem izmed zagonov je algoritem bodisi dosegel ciljno kakovost bodisi prenehal z izvajanjem, ker so bili izpolnjeni pogoji za ustavitev (pri simuliranem ohlajanju dosežena končna temperatura, pri ostalih dveh algoritmih doseženo največje število iteracij brez izboljšave kakovosti rešitve).

Pri algoritmu simulirano ohlajanje se za iteracijo šteje določeno število znižanj temperature, in sicer 500, saj je tako določeno s parametri pri vzporednem algoritmu. Za zaporedni algoritem uporabimo isti kriterij, da lahko rezultate med seboj primerjamo. To lahko ponazorimo na primeru: če je algoritem opravil 3000 znižanj temperature, se to odrazi kot 6 iteracij. Pri ostalih dveh algoritmih štejemo vsako iteracijo algoritma kot dejansko iteracijo.

Pri profilu algoritma simulirano ohlajanje, slika 7.1, lahko ugotovimo, da je vzporedni algoritem v večini primerov hitreje prišel do enako dobre ali boljše rešitve. Poleg tega lahko opazujemo še razliko med uporabo štirih in 16 procesov. V splošnem uporaba 16 procesov povzroči, da algoritem hitreje pride do enako dobre ali boljše rešitve. Problem se je pojavil pri primerku MANN_a81, kjer je bil vzporedni algoritem slabši od zaporednega. Očitno se je pri tem primerku v večini primerov zgodilo, da je vzporedni algoritem obtičal v globokem lokalnem optimumu, iz katerega se ni mogel več rešiti. Zanimivo je izpostaviti še primerek keller6, pri katerem se vzporedni algoritem, ki uporablja štiri procese, obnaša precej podobno kot zaporedni algoritem.

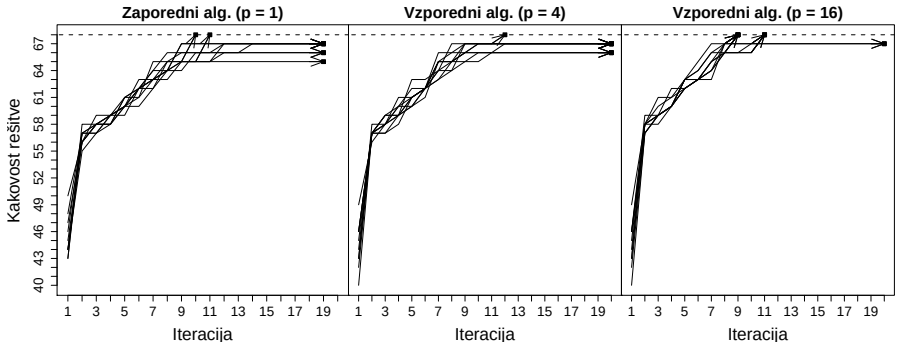
Slika 7.1: Profil izvajanja simuliranega ohlajanja

(a) Primerek brock200_1, ciljna kakovost rešitve = 21

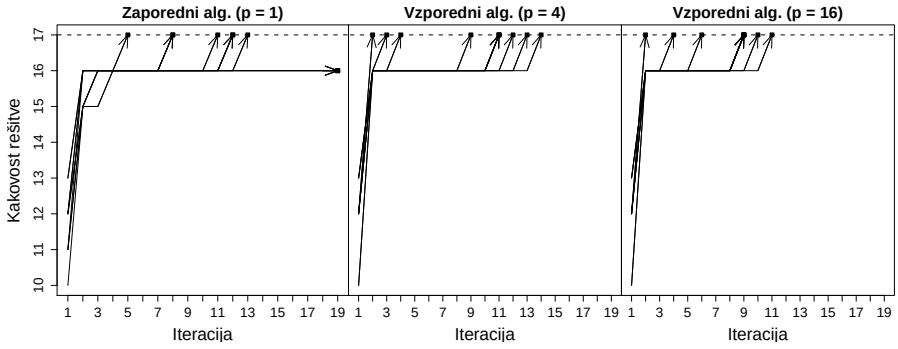


Slika 7.1: Profil izvajanja simuliranega ohlajanja (nadaljevanje)

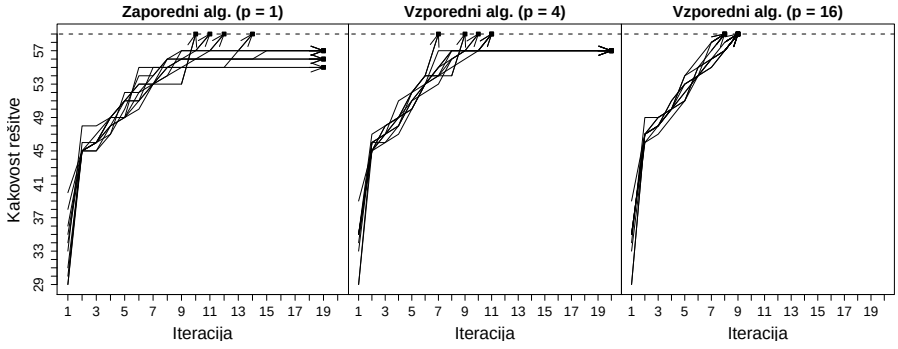
(b) Primerek C1000.5, ciljna kakovost rešitve = 68



(c) Primerek C4000.5, ciljna kakovost rešitve = 17

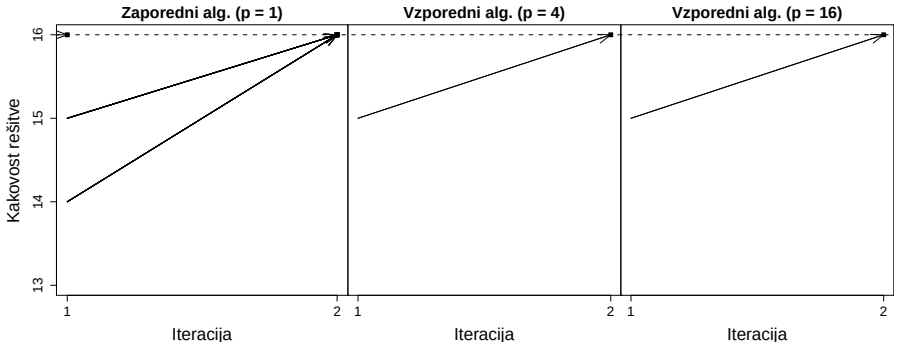


(d) Primerek keller6, ciljna kakovost rešitve = 59

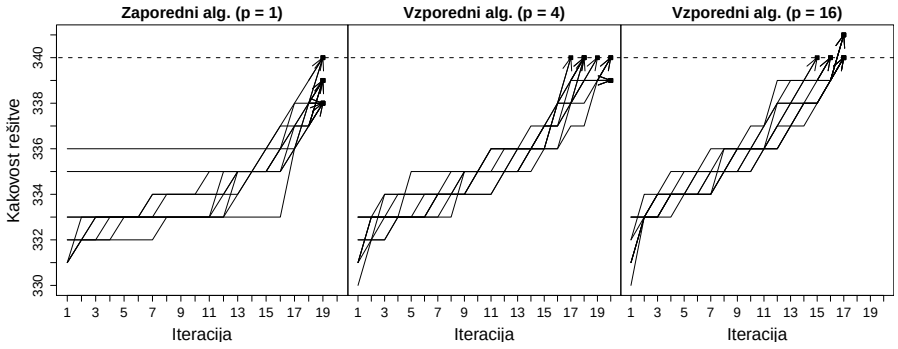


Slika 7.1: Profil izvajanja simuliranega ohlajanja (nadaljevanje)

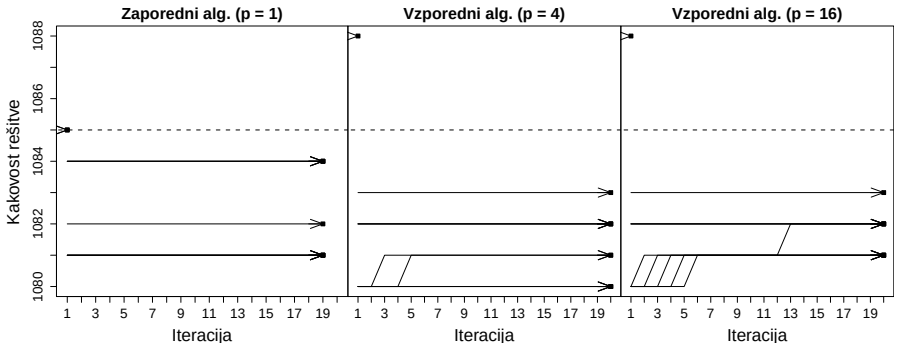
(e) Primerek MANN_a9, ciljna kakovost rešitve = 16



(f) Primerek MANN_a45, ciljna kakovost rešitve = 340



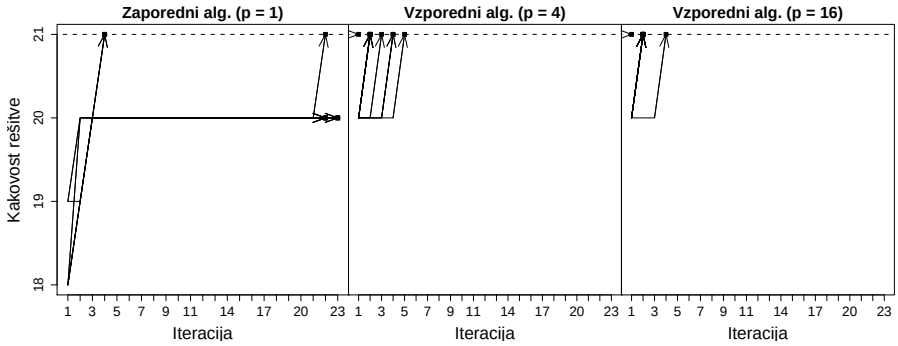
(g) Primerek MANN_a81, ciljna kakovost rešitve = 1085



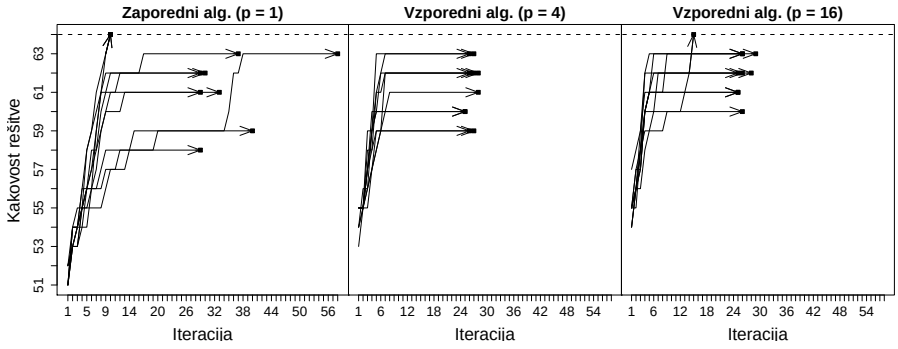
Pri profilu algoritma razpršeno iskanje, slika 7.2, opazimo podobno kot pri simuliranem ohlajanju, da je vzporedni algoritem prišel do enako dobre ali boljše rešitve pri vseh primerkih. Zanimivo je pri primerku keller6 opaziti podoben pojav kot pri simuliranem ohlajanju, in sicer da se vzporedni algoritem, ki uporablja štiri procese, obnaša zelo podobno kot zaporedni algoritem. Kakovost rešitve je pri vzporednem algoritmu s 16 procesi pogosto precej boljša od vzporednega algoritma. To si lahko ponovno razlagamo z dejstvom, da je vzporedni algoritem preiskal večji prostor rešitev, saj se je hkrati izvajal na 16 procesih. Pri štirih procesih pa razlika ni več tako očitna.

Slika 7.2: Profil izvajanja razpršenega iskanja

(a) Primerek brock200_1, ciljna kakovost rešitve = 21

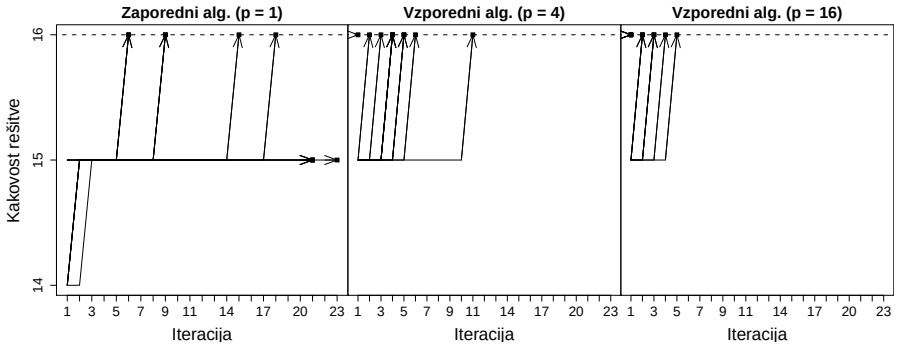


(b) Primerek C1000.5, ciljna kakovost rešitve = 64

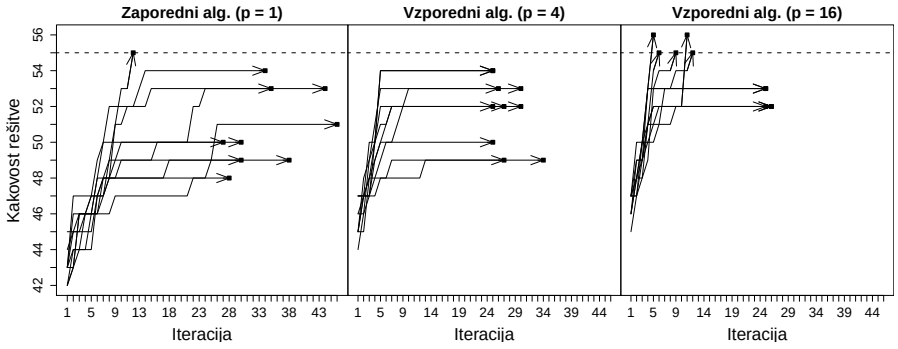


Slika 7.2: Profil izvajanja razpršenega iskanja (nadaljevanje)

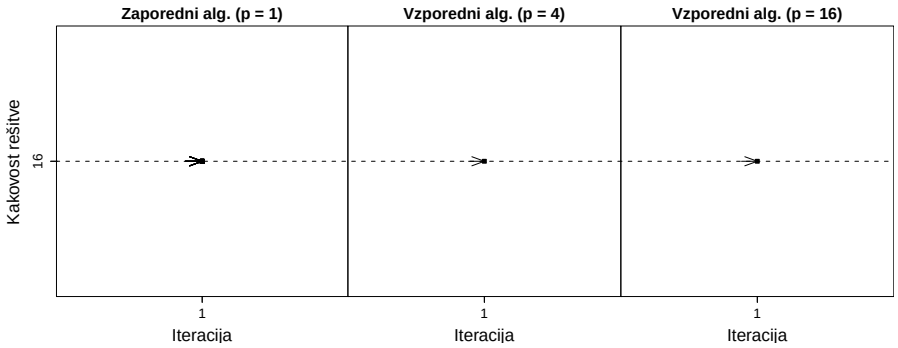
(c) Primerek C4000.5, ciljna kakovost rešitve = 16



(d) Primerek keller6, ciljna kakovost rešitve = 55

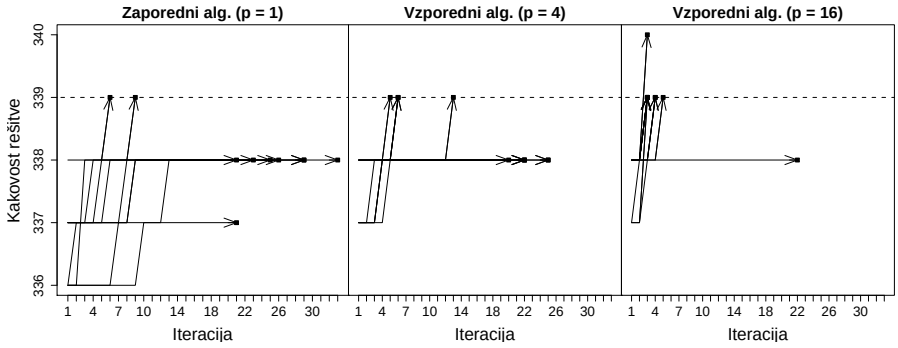


(e) Primerek MANN_a9, ciljna kakovost rešitve = 16

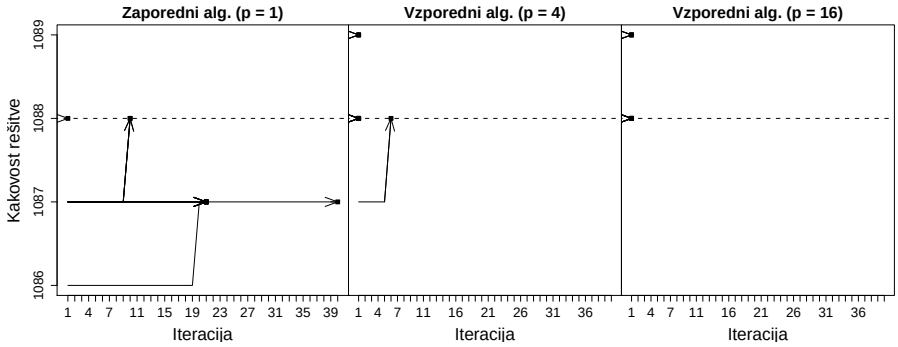


Slika 7.2: Profil izvajanja razpršenega iskanja (nadaljevanje)

(f) Primerek MANN_a45, ciljna kakovost rešitve = 339



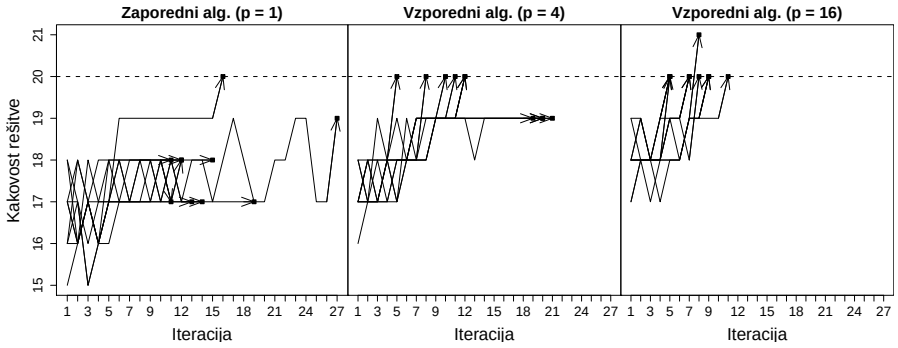
(g) Primerek MANN_a81, ciljna kakovost rešitve = 1088



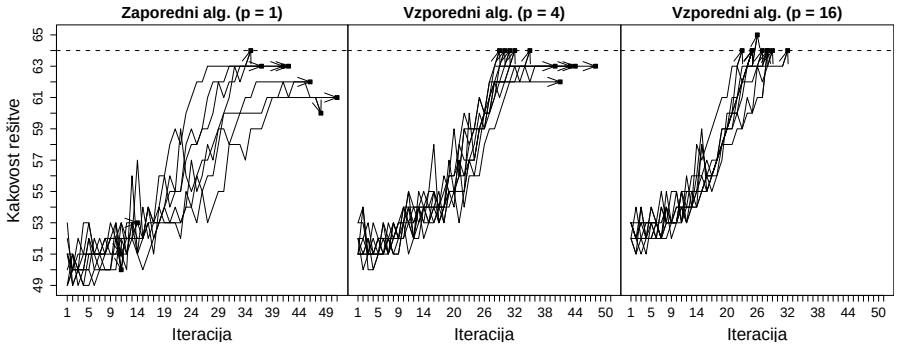
Pri profilu navzkrižne entropije, slika 7.3, lahko ugotovimo, da je v vseh primerih vzporedni algoritem, ki uporablja 16 procesov, hitreje prišel do enako dobre ali boljše rešitve. Pri štirih procesih razlika ni več tako prepričljiva. Če ponovno pogledamo primerek keller6, je imel vzporedni algoritem s štirimi procesi precejšnje težave, tisti s 16 procesi pa ne. Pri slednjem opazimo, da je našel zelo dobre rešitve, vendar je bilo to plačano s precej večjim številom iteracij kot pri zaporednem algoritmu. Iz tega vidika je algoritem razpršenega iskanja na boljšem, saj pri njem ni potrebno plačati te cene. Dodajmo še, da se pri večini ostalih primerkov algoritem, ki uporablja štiri procese, ali bolj nagiba k zaporednemu algoritmu ali bolj k vzporednemu algoritmu, odvisno od primerka.

Slika 7.3: Profil izvajanja navzkrižne entropije

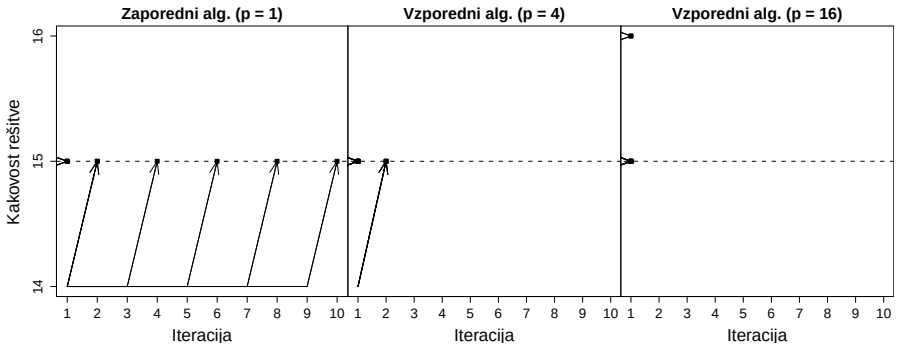
(a) Primerek brock200_1, ciljna kakovost rešitve = 20



(b) Primerek C1000.5, ciljna kakovost rešitve = 64

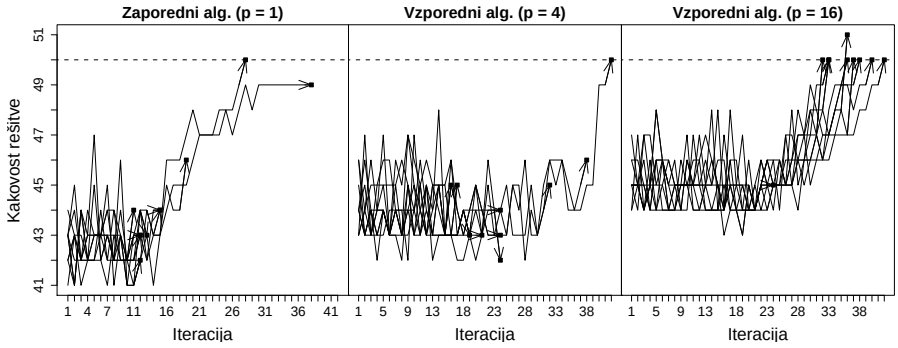


(c) Primerek C4000.5, ciljna kakovost rešitve = 15

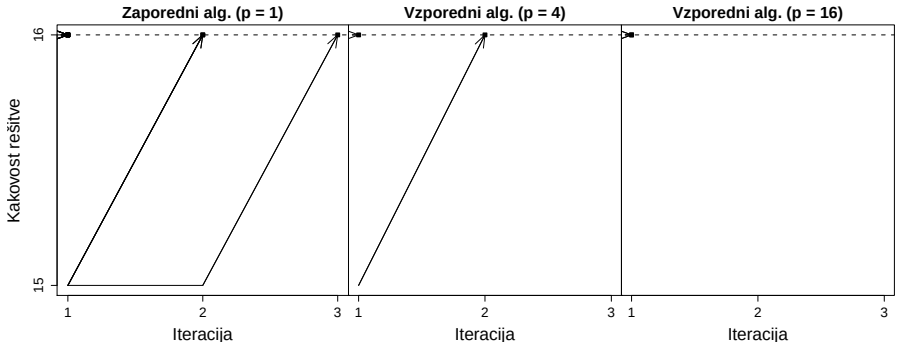


Slika 7.3: Profil izvajanja navzkrižne entropije (nadaljevanje)

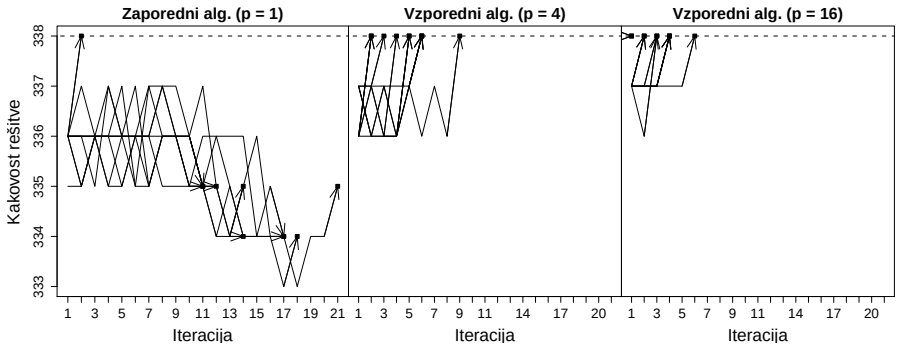
(d) Primerek keller6, ciljna kakovost rešitve = 50



(e) Primerek MANN_a9, ciljna kakovost rešitve = 16

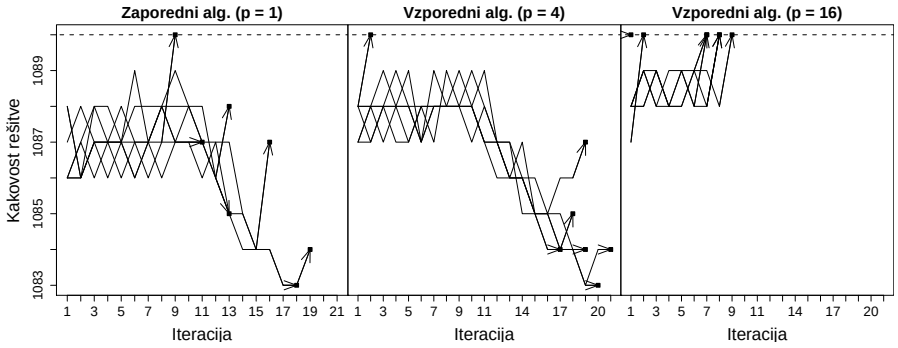


(f) Primerek MANN_a45, ciljna kakovost rešitve = 338



Slika 7.3: Profil izvajanja navzkrižne entropije (nadaljevanje)

(g) Primerek MANN_a81, ciljna kakovost rešitve = 1090



Opazovanje profilov kakovosti rešitev skozi iteracije zaključimo z ugotovitvijo, da je v določenih primerih povzporedenje določenega zaporednega algoritma po eni strani povzročilo doseganje boljših rešitev, po drugi strani pa je v večini primerov povzporedenje povzročilo hitrejšo konvergenco algoritma. Obe izboljšavi nista nujno hkratni in lahko se zgodi, da je konvergenca prehitra in je zaradi tega kakovost rešitve slabša.

Ugotovimo lahko še, da se je s povzporedenjem očitno povečala izostritev, vendar razpršitev temu ni v zadostni meri sledila. To je imelo za posledico omenjeno hitrejšo konvergenco. Izpostaviti moramo še, da smo opazili, da je vzporedni algoritem z uporabo štirih procesov nekakšna mešanica zaporednega algoritma in vzporednega algoritma, ki uporablja 16 procesov. Odvisno od primerka se bolj nagiba v eno ali drugo stran. V veliki večini primerov je bil vzporedni algoritem, ki uporablja 16 procesov, uspešnejši pri doseganju boljših rešitev in pogosto v manjšem številu iteracij kot vzporedni algoritem s štirimi procesi. To nakazuje na dobro plat prikazanega povzporejanja, saj si z večanjem števila procesov vedno želimo izboljšavo – najsibo v večji kakovosti rešitve ali v manjšem številu iteracij, ki ima za posledico krajši čas izvajanja.

Poglavje 8

Zaključek in nadaljnje delo

8.1 Zaključek

V pričujočem delu smo si pogledali določene težke probleme in se v tem okviru seznanili s teorijo NP-polnosti. Natanko smo definirali pojme, ki opredeljujejo, kako težki so določeni problemi in si ogledali štiri NP-ekvivalentne probleme.

Nadaljevali smo s pregledom dela področja kombinatorične optimizacije, in sicer smo se seznanili z metahevristikami. Z njimi rešujemo probleme, za katere ne poznamo algoritmov, ki bi zahtevali čas, krajši od eksponentnega. Če nam zadoščajo suboptimalne rešitve, lahko takšno rešitev dobimo bistveno hitreje kot v eksponentnem času. Ogledali smo si šest metahevrističnih algoritmov: požrešno iskanje, simulirano ohlajanje, razpršeno iskanje, metoda navzkrižne entropije, evolucijski algoritem in sistem mravelj. Izpostavili smo principa razpršitve in izostritve. Prvi govori o širokem preiskovanju prostora rešitev in drugi o natančnem preiskovanju manjšega dela prostora rešitev.

Sledil je opis različnih orodij, ki se uporabljajo za pripravo vzporednih algoritmov, tako na manjših sistemih kot na superračunalnikih. Podrobno smo si ogledali PVM in eno izmed implementacij vmesnika MPI, OpenMPI.

Z željo po pridobitvi rešitve v čim krajšem času smo si pogledali pripravo vzporednih različic treh metahevrističnih algoritmov: simulirano ohlajanje, razpršeno iskanje in metoda navzkrižne entropije. Za osnovo smo vzeli zaporedne algoritme in jih ustrezno predelali, da jih je bilo mogoče hkrati izvajati kot več procesov, pri čemer so si ti procesi izmenjevali informacije o dotlej najdenih rešitvah.

Ker smo želeli zaporedne in vzporedne algoritme med seboj empirično primerjati, smo si najprej ogledali sisteme, na katerih smo kasneje izvedli preizkuse. Pri tem smo vsakega od sistemov natančno opisali in podali informacije

o dolžini izvajanja primerjalnega preizkusa DIMACS.

V predzadnjem poglavju smo si ogledali empirične rezultate treh zaporednih in vzporednih metahevrističnih algoritmov za reševanje problema največje neodvisne množice. Rezultati so bili pridobljeni na opisanih sistemih z uporabo DIMACS primerkov.

Za predstavljeni zaporedni algoritem simuliranega ohlajanja za reševanje problema največje neodvisne množice vozlišč na grafu smo pokazali, da je v splošnem boljši od najboljšega znanega algoritma. Pri 35 primerkih od 80 je predstavljeni algoritem dosegel boljši rezultat in slabšega le pri dveh primerkih.

Na podlagi empiričnih rezultatov smo ugotovili, da se predstavljene vzporedne različice treh metahevrističnih algoritmov ne obnesejo preveč dobro. Opazili smo določene prednosti vzporednih algoritmov z vidika boljše kakovosti rešitve. Vendar nismo videli pohitritve, ki jo lahko sicer upravičeno pričakujemo pri povzporejanju. Rezultat smo v večji meri pripisali večjemu poudarku na izostritvi kot na razpršitvi, kar je bila posledica povzporejanja.

Eden izmed zanimivih rezultatov je bil, da se je z vidika časa izvajanja PVM obnesel bolje od OpenMPI pri algoritmih simulirano ohlajanje in razpršeno iskanje in to kljub temu, da je imel OpenMPI na razpolago hitro omrežje InfiniBand.

8.2 Nadaljnje delo

Za primerjavo rezultatov bi bilo zanimivo pripraviti vzporedne algoritme z uporabo naivnega povzporejanja, jih empirično ovrednotiti in primerjati s predlaganimi tremi vzporednimi algoritmi. Lahko bi se izkazalo, da se kompleksno povzporejanje, ki smo si ga ogledali, odreže slabše in ne prinese pričakovanega rezultata. V takem primeru bi zaključili, da tovrstno povzporejanje ni smiselno (obstaja še možnost, da bi drugačen način kompleksnega povzporejanja prinesel boljše rezultate).

Menimo, da je neprepričljiv rezultat vzporednih algoritmov rezultat slabe razpršitve. Kot smo na več mestih opazili, so vzporedni algoritmi relativno hitro konvergirali, kar je posledica prevelike izostritve in pomanjkanja razpršitve. Če bi konvergenco upočasnili, bi verjetno dosegli boljše rezultate. Zato je tukaj verjetno še precej prostora za izboljšave.

Poleg tega je pri simuliranem ohlajanju in razpršenem iskanju odprta možnost izboljšave lokalnega iskanja [44]. Pri tem pa je seveda potrebno paziti na to, da se preveč ne poveča izostritev, ki je že tako velika, vsaj ne brez bistvenega povečanja razpršitve kot protiuteži.

Kot že omenjeno pri primerjavi vzporednih algoritmov, bi bilo smiselno preizkusiti hitrejšo implementacijo skupinske operacije All_reduce pri PVM. Morda bi se izkazalo, da je PVM komunikacijsko boljši od OpenMPI ne samo pri simuliranem ohlajanju in razpršenem iskanju, temveč tudi pri metodi navzkrižne entropije. Poleg tega bi bilo zanimivo pri primerjavi med OpenMPI in PVM z raznolikimi preizkusi ugotoviti, pod katerimi pogoji se PVM obnese boljše kot OpenMPI.

Eden izmed zanimivih preizkusov bi bil, kako se vzporedni algoritmi obnašajo, ko se število procesov poveča na 32, 64 ali celo še več. Na ta način bi lahko ugotovili, v kakšni smeri se s povečevanjem števila procesov razvija razmerje med razpršitvijo in izostritvijo.

Odprta je še možnost izboljšave rezultatov opisanih zaporednih in vzporednih algoritmov s spreminjanjem parametrov algoritmov. Zelo verjetno obstaja kombinacija parametrov, ki da boljše rezultate od tistih, ki smo si jih ogledali. Res je tudi, da je tovrstno natančno nastavljanje parametrov relativno zamudno opravilo.

Dodatek A

Namestitev vzporednih okolij

OpenMPI 1.3.3 na FreeBSD 6.2 in 7.0 preko zbirke „ports“

```
# Jave ne potrebujemo
root$ export WITHOUT_JAVA=yes
root$ cd /usr/ports/net/openmpi
# Namestitev
root$ make install clean
# Nastavitev spremenljivk (za uporabnike OpenMPI)
root$ echo "export PATH=/usr/local/mpi/openmpi/bin:$PATH" >> \
    ~rokcvahte/.bashrc
```

OpenMPI 1.3.3 na Ubuntu 7.10 in 8.10 preko izvorne kode

```
# Prenos izvorne kode
root$ cd ~
root$ wget http://www.open-mpi.org/software/ompi/v1.3/downloads/\
    openmpi-1.3.3.tar.gz
root$ tar -xzf openmpi-1.3.3.tar.gz
root$ cd openmpi-1.3.3
# Izbira namestitvenega direktorija
root$ ./configure --prefix=/usr/local/openmpi-1.3.3
# Namestitev
root$ make all install
# Nastavitev spremenljivk (za uporabnike OpenMPI)
root$ echo "export PATH=/usr/local/openmpi-1.3.3/bin:$PATH" >> \
    ~rokcvahte/.bashrc
root$ echo "export LD_LIBRARY_PATH=/usr/local/openmpi-1.3.3/lib" >> \
    ~rokcvahte/.bashrc
```

OpenMPI 1.3 na Ubuntu 9.04 preko orodja „APT“

```
root$ apt-get install libopenmpi-dev openmpi-bin openmpi-doc ssh
```

OpenMPI 1.2.6 na CentOS 4.6 preko izvorne kode

```
# Prenos izvorne kode
root$ cd ~
root$ wget http://www.open-mpi.org/software/ompi/v1.2/downloads/\
    openmpi-1.2.6.tar.bz2
root$ tar -xvjf openmpi-1.2.6.tar.bz2
root$ cd openmpi-1.2.6
# Izbira namestitvenega direktorija
root$ ./configure --prefix=/usr/local/openmpi-1.2.6
# Namestitev
root$ make all install
# Nastavitev spremenljivk (za uporabnike OpenMPI)
root$ echo "export PATH=/usr/local/openmpi-1.2.6/bin:$PATH" >> \
    ~rokcvahte/.bashrc
root$ echo "export LD_LIBRARY_PATH=/usr/local/openmpi-1.2.6/lib" >> \
    ~rokcvahte/.bashrc
```

PVM 3.4.5 na FreeBSD 6.2 in 7.0 preko zbirke „ports“

```
# Jave ne potrebujemo
root$ export WITHOUT_JAVA=yes
root$ cd /usr/ports/net/pvm
# Namestitev
root$ make install clean
# Nastavitev spremenljivk (za uporabnike PVM)
root$ echo "export PVM_ROOT=/usr/local/lib/pvm" >> ~rokcvahte/.bashrc
root$ echo "export PVM_RSH=/usr/bin/ssh" >> ~rokcvahte/.bashrc
```

PVM 3.4.5 na Ubuntu 8.10 in 9.04 preko orodja „APT“

```
root$ apt-get install pvm pvm-dev
# Nastavitev spremenljivk (za uporabnike PVM)
root$ echo "export PVM_ROOT=/usr/lib/pvm3" >> ~rokcvahte/.bashrc
```

PVM 3.4.6 na Ubuntu 9.04 preko izvorne kode

```
root$ export PVM_ROOT=/usr/local/pvm
root$ apt-get install m4
# Prenos izvorne kode
root$ cd ~
root$ wget http://www.netlib.org/pvm3/pvm3.4.6.tgz
root$ tar -xvf pvm3.4.6.tgz
root$ mv pvm3 /usr/local/pvm
root$ cd /usr/local/pvm
# Namestitev
root$ make
# Popravek zaradi napake "gs_getgstid() failed to start group server: No\
    such file"
root$ ln -s /usr/bin/pvmgs /home/rok/pvm3/bin/LINUX/pvmgs
# Nastavitev spremenljivk (za uporabnike PVM)
root$ echo "export PVM_ROOT=/usr/local/pvm" >> ~rokcvahte/.bashrc
```

PVM 3.4.6 na CentOS 4.6 preko izvorne kode

```
root$ export PVM_ROOT=/usr/local/pvm
# Prenos izvorne kode
root$ wget http://www.netlib.org/pvm3/pvm3.4.6.tgz
root$ tar -xvf pvm3.4.6.tgz
root$ mv pvm3 /usr/local/pvm
root$ cd /usr/local/pvm
# Namestitev
root$ make
# Nastavitev spremenljivk (za uporabnike PVM)
root$ echo "export PVM_ROOT=/usr/local/pvm" >> ~rokcvahte/.bashrc
root$ echo "export PATH=$PVM_ROOT/lib/:$PATH" >> ~rokcvahte/.bashrc
```

Dodatek B

Prevajanje in zagon vzporednih programov

OpenMPI prevajanje na FreeBSD in Ubuntu

```
rokcvahte$ mpicc izvornaDatoteka.cpp -o izvrsljivaDatoteka
```

OpenMPI zagon

```
# Zagon štirih procesov izvršljive datoteke (mpi_hostfile vsebuje spisek
# računalnikov)
rokcvahte$ mpirun -n 4 -hostfile mpi_hostfile izvrsljivaDatoteka
```

PVM prevajanje na Ubuntu

```
rokcvahte$ gcc -O3 -lpvm3 -lgpvm3 izvornaDatoteka.cpp -o izvrsljivaDatoteka
```

PVM prevajanje na FreeBSD

```
rokcvahte$ gcc -O3 -L/usr/local/lib -lpvm3 -lgpvm3 izvornaDatoteka.cpp -o \
    izvrsljivaDatoteka
```

PVM zagon

```
# Prekopiraj izvršljivo datoteko v privzeti direktorij za PVM
rokcvahte$ cp izvrsljivaDatoteka ~/pvm/bin/$PVM_ARCH/
# Zagon PVM okolja (pvm_hostfile vsebuje spisek računalnikov)
rokcvahte$ pvm pvm_hostfile
# Zagon štirih procesov izvršljive datoteke (znotraj PVM)
pvm> spawn -> -4 izvrsljivaDatoteka
```

Algoritmi

Slike

Tabele

Literatura

[1] (2010, jan.) Islovar. Dostopno na: http://www.islovar.org

[2] (2010, jan.) Slovarček krajšav. Dostopno na: http://bos.zrc-sazu.si/kratice.html

[3] M. R. Garey in D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman & Co., 1979.

[4] A. Grama, G. Karypis, V. Kumar, in A. Gupta, *Introduction to Parallel Computing*, 2. izd. Addison Wesley, 2003.

[5] C. Blum in A. Roli, „Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys*, zv. 35, št. 3, str. 268–308, sept. 2003.

[6] S. A. Cook, „The complexity of theorem-proving procedures," v *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*. ACM Press, 1971, str. 151–158.

[7] W. I. Gasarch, „The P=?NP poll," *ACM SIGACT News*, zv. 33, str. 34–47, jun. 2002.

[8] A. J. Menezes, P. C. van Oorschot, in S. A. Vanstone, *Handbook of Applied Cryptography (Discrete Mathematics and Its Applications)*, 1. izd. CRC Press, 1996.

[9] R. M. Karp, „Reducibility among combinatorial problems," v *Complexity of Computer Computations*, R. E. Miller in J. W. Thatcher, ured. Plenum Press, 1972, str. 85–103.

[10] H. H. Hoos in T. Stützle, *Stochastic Local Search: Foundations & Applications*, 1. izd. Morgan Kaufmann, 2004.

[11] S. Kirkpatrick, C. D. Gelatt, Jr, in M. P. Vecchi, „Optimization by simulated annealing,“ *Science*, zv. 220, str. 671–680, maj 1983.

[12] D. Henderson, S. H. Jacobson, in A. W. Johnson, „The theory and practice of simulated annealing,“ v *Handbook of Metaheuristics*, F. W. Glover in G. A. Kochenberger, ured. Springer, 2003, str. 287–319.

[13] E. Aarts, J. Korst, in W. Michiels, „Simulated annealing,“ v *Handbook of Approximation Algorithms and Metaheuristics*, T. F. Gonzalez, ured. Chapman & Hall/CRC, 2007, pogl. 25.

[14] X. Geng, J. Xu, J. Xiao, in L. Pan, „A simple simulated annealing algorithm for the maximum clique problem,“ *Information Sciences*, zv. 177, št. 22, str. 5064–5071, nov. 2007.

[15] X. Xu in J. Ma, „An efficient simulated annealing algorithm for the minimum vertex cover problem,“ *Neurocomputing*, zv. 69, št. 7-9, str. 913–916, mar. 2006.

[16] R. Martí, M. Laguna, in F. Glover, „Principles of scatter search,“ *European Journal of Operational Research*, zv. 169, št. 2, str. 359–372, mar. 2006.

[17] M. Laguna in R. Martí, *Scatter Search*. Springer, 2003.

[18] L. Cavique, C. Rego, in I. Themido, „A scatter search algorithm for the maximum clique problem,“ Instituto Politecnico de Lisboa, Portugal, teh. poroč. HCES-01-01, jan. 2001.

[19] P. T. De Boer, D. P. Kroese, S. Mannor, in R. Y. Rubinstein, „A tutorial on the cross-entropy method,“ *Annals of Operations Research*, zv. 134, št. 1, str. 19–67, feb. 2005.

[20] G. Alexe, G. Bhanot, in A. Climescu-haulica, „A cross entropy algorithm for classification with δ–patterns,“ v *DMTCS proceedings AG*, ser. Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities, 2006, str. 399–402.

[21] Q. Lü, Z.-H. Bai, in X.-Y. Xia, „Leader-based parallel cross entropy algorithm for maximum clique problem,“ *Journal of Software*, zv. 19, št. 11, str. 2899–2907, apr. 2009.

[22] C. Reeves, „Genetic algorithms,“ v *Handbook of Metaheuristics*, F. W. Glover in G. A. Kochenberger, ured. Springer, 2003, str. 55–82.

[23] G. Leguizamón, C. Blum, in E. Alba, „Evolutionary computation,“ v *Handbook of Approximation Algorithms and Metaheuristics*, T. F. Gonzalez, ured. Chapman & Hall/CRC, 2007, pogl. 24.

[24] K. Deb, „An efficient constraint handling method for genetic algorithms,“ *Computer Methods in Applied Mechanics and Engineering*, zv. 186, št. 2-4, str. 311–338, jun. 2000.

[25] M. Dorigo in T. Stützle, *Ant Colony Optimization*. The MIT Press, 2004.

[26] W. Gutjahr, „Mathematical runtime analysis of ACO algorithms: survey on an emerging issue,“ *Swarm Intelligence*, zv. 1, št. 1, str. 59–79, jun. 2007.

[27] W. Feng in P. Balaji, „Tools and environments for multicore and many-core architectures,“ *Computer*, zv. 42, št. 12, str. 26–27, dec. 2009.

[28] (2009, sept.) MPI documents. Dostopno na: http://www.mpi-forum.org/docs/docs.html

[29] A. Geist, A. Beguelin, J. Dongarra, W. Jiang *in sod.*, *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT Press, 1994.

[30] B. Jacob, M. Brown, K. Fukui, in N. Trivedi, *Introduction to Grid Computing*, ser. IBM Redbooks. IBM Corp., 2005.

[31] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis *in sod.*, „Cloud computing: Distributed internet computing for it and scientific research,“ *IEEE Internet Computing*, zv. 13, št. 5, str. 10–13, sept. 2009.

[32] L. Ferreira, V. Berstis, J. Armstrong, M. Kendzierski *in sod.*, *Introduction to Grid Computing with Globus*, ser. IBM Redbooks. IBM Corp., 2003.

[33] (2010, jan.) OpenMPI F.A.Q. Dostopno na: http://www.open-mpi.org/faq/

[34] (2009, feb.) PVM release notes. Dostopno na: http://www.netlib.org/pvm3/RELEASE_NOTES.txt

[35] I. Krüger, W. Prenninger, R. Sandner, in R. S, „Deriving architectural prototypes for a broadcasting system using UML-RT,“ v *Proceedings 1st ICSE Workshop on Describing Software Architecture with UML*, P. Kruchten, ured., 2001.

[36] O. So in L. Özdamar, „Parallel simulated annealing algorithms in global optimization,“ *Journal of Global Optimization*, zv. 19, str. 27–50, jan. 2001.

[37] G. Kliewer in S. Tschoke, „A general parallel simulated annealing library and its application in airline industry,“ v *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000.* IEEE Computer Society, 2000, str. 55–61.

[38] C. D. Gijo, E. Polite, in E. Te, „An empirical investigation on parallelization strategies for scatter search,“ *European Journal Of Operational Research*, zv. 169, str. 490–507, mar. 2006.

[39] G. E. Evans, J. M. Keith, in D. Kroese, „Parallel cross-entropy optimization,“ v *WSC '07: Proceedings of the 39th conference on Winter simulation.* IEEE Press, 2007, str. 2196–2202.

[40] (2008, jun.) TOP500 supercomputing sites – TOP500 list. Dostopno na: http://www.top500.org/list/2008/06/100

[41] D. S. Johnson in M. A. Trick, „Cliques, coloring, and satisfiability: Second DIMACS implementation challenge,“ v *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, okt. 1993, zv. 26, str. 1–7.

[42] (1993, sept.) DIMACS instances. Dostopno na: ftp://dimacs.rutgers.edu/pub/challenge

[43] W. Pullan in H. H. Hoos, „Dynamic local search for the maximum clique problem,“ *Journal of Artificial Intelligence Research*, zv. 25, št. 1, str. 159–185, jan. 2006.

[44] T. A. El-Mihoub, A. A. Hopgood, L. Nolle, in A. Battersby, „Hybrid genetic algorithms – a review,“ *Engineering Letters*, zv. 13, št. 2, str. 124–137, avg. 2006.